



Genis Bonet Garcia

Developing a Reduced 2D Model for Rotating Detonation Engine
Simulations

Faculty of Engineering and Applied Science
Astronautics and Space Engineering

MSc
Academic Year: 2024 - 2025

Supervisor: Enric Grustan
September 2025



Faculty of Engineering and Applied Science
Astronautics and Space Engineering

MSc

Academic Year 2024 - 2025

Genis Bonet Garcia

Developing a Reduced 2D Model for Rotating Detonation Engine
Simulations

Supervisor: Enric Grustan
September 2025

This thesis is submitted in partial fulfilment of the requirements for
the degree of Astronautics and Space Engineering

© Cranfield University 2025. All rights reserved. No part of this
publication may be reproduced without the written permission of the
copyright owner.

Academic integrity declaration

I declare that:

- the thesis submitted has been written by me alone.
- the thesis submitted has not been previously submitted to this university or any other.
- that all content, including primary and/or secondary data, is true to the best of my knowledge.
- that all quotations and references have been duly acknowledged according to the requirements of academic research.

I understand that to knowingly submit work in violation of the above statement will be considered by examiners as academic misconduct.

Abstract

This thesis develops a compact, reproducible two-dimensional finite-volume Euler solver in vectorized Python to support exploratory detonation studies. The flow solver uses MUSCL reconstruction with an HLLE approximate Riemann flux and a strong-stability-preserving Runge–Kutta time integrator, emphasizing portability and clear, testable software structure. Verification exercises include a quantitative Sod shock tube study against an exact Riemann solution and a qualitative circular blast symmetry check; additional timing sweeps compare a MATLAB baseline to the vectorized Python port. Thermochemistry is designed but not exercised in the CFD results: instead, a standalone JAX-based multi-reactor module is validated against Cantera and benchmarked for scaling, demonstrating increasing speedups with problem size while maintaining parity in instantaneous rates and thermodynamic conversions. Together, the results indicate close agreement with the MATLAB reference on canonical tests and credible throughput gains for batched chemistry, with an explicit path to operator-split coupling. Limitations include the absence of in-line chemistry in the delivered solver and a focus on ideal-gas mixtures; future work targets robust operator-split integration and extended verification and validation.

Keywords:

Finite-volume methods, MUSCL, HLLE, operator splitting, Euler equations, Arrhenius, NASA-7, JAX, Cantera, Optimization

Acknowledgements

I would like to express my most sincere gratitude to my supervisor, **Enric Grustan**, for his guidance, encouragement, and patience throughout this project. His steady feedback shaped the direction of the work and helped me maintain focus when the scope grew challenging.

I am also grateful to **Reuben Hann**, Cranfield University alumnus, for generous discussions and practical advice during development. Parts of this thesis build, loosely and with attribution, on his adaptations of the original MATLAB codebase, which I subsequently ported and extended.

Table of Contents

Academic integrity declaration.....	i
Abstract	ii
Acknowledgements	iii
List of Figures.....	vii
List of Tables	ix
List of Equations.....	x
List of Abbreviations	xii
1 Introduction.....	1
1.1 Motivation	1
1.2 Numerical & Software Rationale	2
1.3 Porting & Performance Engineering.....	3
1.4 Aims and Scope.....	3
1.5 Contributions.....	4
2 Background & Related Work	5
2.1 Detonation waves and rotating detonation engines	5
2.2 Shock-capturing methods for compressible flow.....	5
2.3 Operator splitting and reactive flow coupling strategies	6
2.4 Thermochemical data and detailed vs. reduced kinetics.....	7
2.5 Verification and (out-of-scope) validation practices	7
2.6 Positioning of this work	8
3 Governing equations	9
3.1 State variables and assumptions	9
3.2 Conservation laws for the target model.....	10
3.3 Detailed chemical kinetics.....	11
3.4 Thermodynamic closure and derived quantities.....	13
3.5 Model scope	13
4 Numerical methods	15
4.1 Finite-volume discretization & notation	15
4.2 MUSCL reconstruction and slope limiting	15
4.3 Numerical flux: HLLC with Einfeldt speeds	16
4.4 Time integration and CFL policy	17
4.5 Physical admissibility and safeguards.....	18
4.6 Per-time-step algorithm (advective part)	18
4.7 Final chemistry RHS.....	18
5 Software design & implementation	20
5.1 Platform choices	20
5.2 Porting strategy.....	21
5.2.1 Baseline reproduction for functional parity	21
5.2.2 Why “array-of-structures” was discarded in Python.....	22
5.2.3 Baseline discrepancy on 2-D Sod (edge and corner artifacts)	23

5.2.4 AoS to SoA transformation in the solver.....	26
5.3 Data layout & vectorization	27
5.3.1 Motivation for vectorization.....	27
5.3.2 Slope reconstruction and limiters (vectorized form)	27
5.3.3 Vectorized flux evaluation	28
5.3.4 Broadcasting defect and resolution	29
5.3.5 Resulting design	29
5.4 Ghost-Cell Conventions and Off-by-One Errors.....	29
5.4.1 Indexing and shapes (1 ghost layer per side).....	30
5.4.2 Ghost-cell fill.....	30
5.5 Debugging methodology	31
5.6 Performance Engineering	33
5.7 Orchestration & Object-Oriented Design.....	34
6 Thermochemistry.....	36
6.1 Scope note.....	36
6.2 Prototype A: Cantera ReactorNet	36
6.3 Prototype B: “rates-only” explicit step	37
6.4 Software implementation of the chemistry RHS.....	37
6.5 Integration plan with CFD	40
7 Verification.....	41
7.1 CFD tests	41
7.1.1 Sod shock tube	41
7.1.2 Non-vectorized circular blast.....	42
7.1.3 Vectorized circular blast	42
7.1.4 Vectorization benchmark harness	43
7.2 Chemistry tests	43
7.2.1 Thermodynamic conversion parity	44
7.2.2 Instantaneous kinetics parity	45
7.2.3 Comprehensive Chemical Kinetics Validation Matrix	46
7.2.4 JAX vs Cantera scaling benchmark	47
7.2.5 Divergence vs initial condition and runtime	47
8 Results	49
8.1 CFD test results	49
8.1.1 Sod shock tube results	49
8.1.2 Circular blast results.....	52
8.1.3 Vectorization benchmark results	53
8.2 Chemistry test results	54
8.2.1 Thermodynamic conversion parity	54
8.2.2 Instantaneous kinetics parity	55
8.2.3 Comprehensive Chemical Kinetics Validation Matrix	56
8.2.4 JAX vs Cantera scaling benchmark	58
8.2.5 Divergence vs initial condition and runtime	62

9 Conclusions.....	67
9.1 CFD solver fidelity from canonical problems.....	67
9.2 Methodological soundness of MUSCL–HLLC–SSP-RK2.....	67
9.3 Performance: CFD solver (vectorization).....	68
9.4 Performance: chemistry integration (JAX vs. Cantera, JIT-compiled).....	68
9.5 Thermochemical layer correctness and independence.....	69
9.6 Where and why numerical trajectories diverge	70
9.7 Scope and limits of what is concluded	71
10 Future work	72
10.1 Reactive coupling at the solver core	72
10.2 Numerical robustness in ignition bands	72
10.3 Performance and scalability.....	72
10.4 Reactive benchmarks and targeted validation	73
References	74
Appendix A Ethical approval letter	81
Appendix B RHS Thermochemistry Landmarks.....	82

List of Figures

Figure 1.1-1 Rotating Detonation Engine from Air Force Research Laboratory.	2
Figure 5.2-1 Original MATLAB solver's array of structures layout.	22
Figure 5.2-2 MATLAB reference solution for the 2-D Sod extrusion.....	24
Figure 5.2-3 Python baseline solution, prior to fixes: edge streaks and corner defects are visible.	24
Figure 5.2-4 Python baseline solution after fixes: single-iteration density results within $e-14$ kg/m ³ tolerance to those obtained by the MATLAB solver. ...	26
Figure 5.2-5 Final Python solver's structures of arrays layout.	27
Figure 5.3-1 Pseudo-code of the vectorized version of Harten's <i>minmod</i> slope limiter. The mask enforces "all candidates same sign"; where signs disagree, the limiter returns zero. Where they agree, it takes the smallest magnitude and re-applies the common sign.	28
Figure 5.3-2 Residual assembly in the x-direction. Left: scalar double-loop over faces; Right: vectorized update where face fluxes and residual contributions are formed and applied over slices in one step.	29
Figure 5.4-1 Ghost cell fill pseudocode.	31
Figure 5.4-2 Reflecting wall ghost cell fill (available for code verification).	31
Figure 5.7-1 Schematic of the implemented CFDGrid object.	34
Figure 6.4-1 Pseudocode of Batched evaluation across a grid: Turns the cell RHS into a batched RHS for N reactors.	38
Figure 6.4-2 Pseudocode of Cell-local RHS: The per-cell derivative $[Y, e]$, enforcing realizability and mass conservation.	39
Figure 6.4-3 Pseudocode of Time integration with Diffrax:	40
Figure 8.1-1 Sod's shock tube results; Density and Energy visual comparisons.	50
Figure 8.1-2 Sod's shock tube results; Velocity and Pressure visual comparisons.	51
Figure 8.1-3 Circular blast results at time 0.19 seconds.....	52
Figure 8.1-4 Vectorization benchmark results; MATLAB; pre-vectorization and post-vectorization execution times alongside vectorization speedup.....	53
Figure 8.2-1 Temperature divergence between JAX and Cantera implementations across varying initial temperatures and integration times at constant pressure (1 atm).	62

Figure 8.2-2 Temperature divergence between JAX and Cantera
implementations across varying initial pressures and integration times at
constant temperature (1500K). 64

B-1 Implemented chemistry mechanism ingestion pseudocode..... 82

B-2 Implemented chemistry thermodynamic closures pseudocode..... 83

B-3 Implemented chemistry third-body falloff pseudocode..... 83

B-4 Implemented chemistry species production rates pseudocode..... 84

B-5 Implemented chemistry equilibrium constants calculation pseudocode..... 84

List of Tables

Table 5.2–1 MATLAB vs Python indexing (with one ghost layer)	25
Table 7.1–1 CFD verification tests with intent, setup and metrics.	41
Table 7.2–1 Chemistry verification tests with intent, setup and metrics.	44
Table 8.1–1 Sod’s shock tube results; MATLAB quantitative divergence.....	49
Table 8.1–2 Sod’s shock tube results; Exact solution quantitative divergence.	50
Table 8.2–1 Thermodynamic conversion parity results (T <-> e).	54
Table 8.2–2 Comprehensive validation matrix comparing JAX and Cantera implementations across multiple temperatures and compositions.....	57
Table 8.2–3 Figure 8.2.4.1: Performance comparison between JAX (purple) and Cantera (desaturated red) implementations showing execution time vs. problem size (left) and speedup vs. problem size (right).	59
Table 8.2–4 Performance Comparison between JAX and Cantera Implementations.	59
Table 8.2–5 Peak Temperature Divergence (%) at Selected Initial Temperatures	63
Table 8.2–6 Peak Temperature Divergence (%) at Selected Initial Pressures.	65

List of Equations

(3-1) Conserved CFD state variables.	9
(3-2) Governing equations (compressible Euler) in conservative form.	10
(3-3) Euler fluxes.	10
(3-4) Ideal gas equations of state.	10
(3-5) Chemical source fluxes.	10
(3-6) Volumetric chemical heat source.	11
(3-7) Net stoichiometric coefficient formula.	11
(3-8) Reaction rate of progress of a stoichiometric reaction.	11
(3-9) Species molar production rate.	11
(3-10) Reduced pressure for falloff reactions.	12
(3-11) Troe blending definition.	12
(3-12) Equilibrium constants from reaction Gibbs energy.	12
(3-13) NASA-7 polynomials for heat capacity, enthalpy and entropy.	13
(4-1) Semi-discrete finite-volume update for the cell-averaged conserved state.	15
(4-2) Minmod definition.	16
(4-3) Component-wise minmod definition.	16
(4-4) Left/right state definition.	16
(4-5) HLLE fluxes.	16
(4-6) Wave-speed bounds with Einfeldt estimates.	17
(4-7) Definitions for Roe-averaged properties.	17
(4-8) HLLE flux definition.	17
(4-9) Strong-stability-preserving (SSP) RK2 update for the hyperbolic operator.	17
(4-10) Global CFL time-step restriction based on local acoustic wave speeds.	17
(4-11) Stoichiometric forward rate constants.	19
(4-12) Species production and heat release rates.	19
(5-1) Semi-discrete update on interior cells.	30

(6-1) Explicit Euler update of species and total energy using evaluations via Cantera.....	37
(6-2) Scaling of source increments by α to enforce realizability.	37

List of Abbreviations

AIAA	American Institute of Aeronautics and Astronautics
AoS/SoA	Array of Structures / Structure of Arrays
BDF	Backward Differentiation Formula
BLAS/ LAPACK/ MKL	Basic Linear Algebra Subprograms/Linear Algebra PACKage/ Intel Math Kernel Library
CFD	Computational fluid dynamics
CFL	Courant–Friedrichs–Lewy (condition)
CJ	Chapman–Jouguet
CLAWPACK	Conservation Laws Package
CPU / GPU	Central Processing Unit / Graphics Processing Unit
CVODES/IDAS	SUNDIALS ODE / DAE solvers
DIRK / ESDIRK	Diagonally Implicit Runge–Kutta / Explicit-first-stage SDIRK
EOS	Equation of state
FVM	Finite-volume method
HLLE / HLLC	Harten–Lax–van Leer–Einfeldt / –Contact (Riemann solvers)
HPC	High performance computing
IC	Initial condition(s)
Isp	Specific impulse
JAX	High-performance autodiff/array library
JIT	Just-in-time (compilation)
MC (limiter)	Monotonized Central slope limiter
NASA / NASA-7	National Aeronautics and Space Administration / 7- coefficient thermochemical polynomials
ODE	Ordinary differential equation
PID	Proportional–Integral–Derivative (controller)
RDE	Rotating detonation engine
RHS	Right-hand side
RK / RK2	Runge–Kutta / 2-stage Heun (SSP-RK2)
SSP	Strong-stability-preserving
SUNDIALS	Suite of Nonlinear and Differential/Algebraic equation Solvers
SSTO	Single stage to orbit

TVD	Total variation diminishing
ZND	Zeldovich–von Neumann–Döring

1 Introduction

Rotating detonation engines (RDEs) use pressure-gain combustion, which obtains heat release through a supersonically traveling detonation that cycles around an annular chamber. In comparison to conventional deflagration, pressure-gain cycles can increase net pressure produced by the combustor and, under fine-tuned circumstances, meaningfully raise specific impulse in relation to conventional designs. The last decade has seen a resurgence of interest in RDEs because they offer a compact and mechanically simple route to higher performance, as far as their unsteady flow physics can be understood and controlled [1], [2], [3].

This thesis aims to develop an open, portable software package for exploratory RDE studies. This package takes the shape of a two-dimensional MUSCL finite-volume solver, ported from MATLAB to vectorized Python, coupled to chemistry via operator splitting. Verification against canonical test problems is used to ensure physical fidelity. The overarching goal is to enable reproducible experiments on detonation wave dynamics with minimal platform friction.

1.1 Motivation

The central motivation of this work is to lower the barrier to exploratory detonation studies by providing a basis for an open and portable simulation tool. High-fidelity RDE simulations are computationally demanding and can require substantial HPC time [4]. A lightweight 2-D framework is valuable as it allows quick testing of idealized operating conditions, early-stage design iteration and accessibility for independent researchers. Positioning the solver in the open enables reproducibility community extension and verification.

In this work, RDEs are used here primarily as a representative modern pressure-gain combustor to motivate the numerical ingredients (shock-capturing, operator splitting, reduced chemistry) and to provide physically meaningful test problems. The last decade's activity around RDEs has been driven by the prospect of cycle-level efficiency gains and well-documented nonidealities that shape realized performance. These characteristics make them

into relevant exemplar without requiring this thesis to advance RDE physics per se. [1], [2].

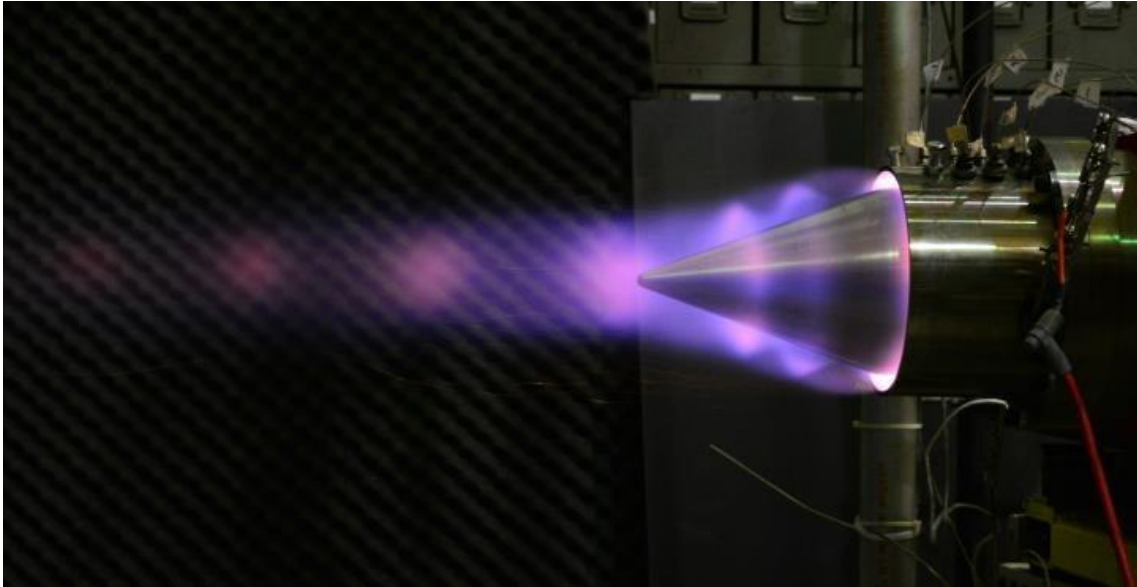


Figure 1.1-1 Rotating Detonation Engine from Air Force Research Laboratory.

Finally, the choice of RDEs against other pressure gain combustion designs is also informed by the forward-thinking design interest of pairing pressure-gain combustors with altitude-compensating nozzles (e.g., aerospike or plug nozzle), see Figure 1.1-1 [3]. These nozzle designs preserve optimal efficiency across any ambient pressure. Although no nozzle coupling is studied here, there is prior work on altitude-compensating RDE concepts. Early studies coupling RDE chambers to plug/aerospike nozzles suggest a promising pathway for future system-level exploration once a rapid, reproducible simulation base is in place [5], [6].

1.2 Numerical & Software Rationale

Computational fluid dynamics (CFD) provides a practical way to study flows relevant to rotating detonation engines without the cost and short and diagnostic limits of physical experiments. For this purpose, the solver uses a two-dimensional finite-volume scheme with MUSCL reconstruction and a HLLC type approximate Riemann solver. Chemistry is coupled by operator splitting so transport and source terms can be advanced with methods suited to each

process; the approach and its second-order Strang variant are well established [7], [8].

To check that numerical results reflect the implemented scheme rather than coding mistakes, code verification includes the Sod shock tube with error norms on multiple grids [9].

1.3 Porting & Performance Engineering

The CFD solver is implemented in Python/NumPy to prioritize portability and reproducibility. Performance work focuses on expressing the update as vectorized array operations so that looping occurs in compiled functions rather than at the Python level. Planned practices include contiguous storage, minimizing temporaries, and a structure-of-arrays layout for the conservative variables. A timing sweep (total wall time vs. problem size) will be used to report relative speedups between the MATLAB (iterative/semi-vectorized) baseline and the fully vectorized Python port; results are reported as overall runtime and speedup, not normalized per grid cell.

Chemistry is prototyped with operator splitting in mind, keeping transport and reactions modular. The near-term plan, as outlined in the IRP notes, is to prototype with per-cell Cantera reactors and then evaluate reduced-chemistry alternatives that are easier to batch and vectorize within NumPy, JAX and Diffrax [10], [11], [12], [13]. This preserves a clear separation between the CFD step (conserved-variable update) and the reaction step (species/internal-energy update) and fits the intended object-oriented orchestration of the workflow.

1.4 Aims and Scope

Aims:

- Provide a 2-D finite-volume solver in Python with MUSCL reconstruction and an HLLE flux, with chemistry coupled via operator splitting and organized for batch/vectorized execution.
- Supply reproducible verification and demonstration cases, including Sod's shock tube with multi-grid error reporting.

In scope:

- MATLAB to Python port with clear documentation of data layout and update steps, emphasizing vectorized array operations.
- Chemical kinetics implementation, first via Cantera reactors and then via chemistry formulations suitable for batched evaluation.
- Code verification on canonical problems (e.g., Sod shock tube).
- Detailed multi-species mechanisms.

Out of scope:

- Three-dimensional chambers or nozzle coupling.
- Large-scale performance engineering (GPU/distributed).

1.5 Contributions

The key contributions of this thesis are:

- A **2-D Python finite-volume solver** (MUSCL + HLLE flux) packaged for reproducible runs.
- A **batched Chemical Kinetics solver** (Arrhenius + NASA-7).
- A **verification suite** (Sod shock tube; multi-grid error reporting) with scripts and configs.
- A **speedup comparison** of MATLAB (iterative/semi-vectorized) vs. NumPy (fully vectorized) implementations over increasing problem sizes.
- A **speedup comparison** of Cantera-based chemistry vs. batched Chemical Kinetics.

2 Background & Related Work

2.1 Detonation waves and rotating detonation engines

Detonation is a supersonic combustion mode in which a thin leading shock compresses and heats the reactive mixture, followed by a finite-rate reaction zone that relaxes to a state satisfying the Chapman–Jouguet (CJ) condition. In the classical one-dimensional Zeldovich–von Neumann–Döring (ZND) picture, the internal structure consists of a lead shock, a high-pressure von Neumann spike in which the gas remains largely unreacted, and a reaction layer whose energy release sustains the shock such that the post-reaction flow becomes sonic in the shock-fixed frame. This framework provides reference quantities, CJ speed, pressure, and an induction length, that are routinely used to organize experiments, assess mechanisms, and benchmark numerical solvers [14].

In rotating detonation engines (RDEs), one or more azimuthally traveling detonation fronts form in an annular combustor, ingesting premixed or mixing reactants, and exhausting through a nozzle. Modern reviews emphasize that RDE operation is dominated by “non-idealities”: cellular detonation structures, shock/boundary-layer interactions, purge and injection dynamics, and multi-cycle unsteadiness that challenge both simplified models and high-fidelity simulations. Understanding and quantifying these effects has motivated a spectrum of modelling choices, from global two-step kinetics in canonical configurations to detailed chemistry in reduced geometries and remains an active area of research [1].

2.2 Shock-capturing methods for compressible flow

The governing equations for inviscid, compressible gas dynamics are hyperbolic conservation laws. Godunov-type finite-volume schemes with approximate Riemann solvers remain the workhorse for shock-dominated problems. The HLLC flux (Harten–Lax–van Leer with Einfeldt’s estimates for signal speeds) is particularly robust because it guarantees positivity of density and pressure under mild conditions and avoids spurious transonic failures near near-vacuum

states; a desirable property for detonation computations with strong rarefactions [15].

Second-order spatial accuracy is commonly obtained with MUSCL reconstruction (Monotone Upstream-centred Scheme for Conservation Laws), which achieves total-variation-diminishing (TVD) behaviour when paired with appropriate slope limiters. The foundational developments by van Leer and the limiter analysis by Sweby (the “Sweby diagram”) provide the theoretical basis for practical limiter choices such as minmod, MC and superbee used to control dispersive oscillations near shocks. Standard texts (e.g., Toro; LeVeque) consolidate these ideas and their implementation details for compressible flows [16], [17].

For time advancement in method-of-lines form, strong-stability-preserving (SSP) Runge–Kutta schemes guarantee that the TVD property of the forward-Euler step is inherited by higher-order updates under a CFL constraint. The widely used two-stage SSP-RK2 (Heun) method offers a favourable balance between cost and stability for shock-capturing codes [18].

2.3 Operator splitting and reactive flow coupling strategies

Reactive compressible simulations frequently exploit operator splitting to separate convection from stiff chemistry. First- and second-order composition via Lie and Strang splitting respectively are standard; the latter provides second-order accuracy for sufficiently regular operators and is routinely used when the reaction time scale is much shorter than the flow time scale. Survey articles from numerical analysis underline accuracy/stability trade-offs and alternatives (e.g., integrating-factor SSP schemes) when stiffness is extreme [7], [8].

In the present work, chemistry is not coupled to the Euler solver; instead, a separate multi-reactor network advanced with JAX-based ODE solvers was prototyped. This decoupled architecture isolates kinetics development from the flow solver and allows independent verification of each subsystem, but it precludes fully reactive CFD results within this thesis scope.

2.4 Thermochemical data and detailed vs. reduced kinetics

Thermodynamic properties of individual species are commonly represented by NASA 7-coefficient polynomials, assembled for thousands of species and validated across broad temperature ranges. The NASA Glenn compilation (McBride, Zehe & Gordon) and the Burcat–Ruscic Third Millennium database remain standard sources in combustion modelling and are widely used by general-purpose libraries such as Cantera and TChem [19], [20].

A central modelling choice is between detailed mechanisms (tens to thousands of species/reactions) and reduced/global mechanisms (few species/reactions tuned for target conditions). Detailed kinetics better capture extinction/ignition limits and pollutant precursors but are stiff and expensive; practical acceleration strategies include analytic/sparse Jacobians (e.g., pyJac) and performance-portable kernels (e.g., TChem). These tools provide verified building blocks for multi-reactor networks and enable rigorous kinetics evaluation without resorting to mechanism reduction [21].

Consistent with that trend, the implementation developed here does not employ a reduced chemical model. Instead, it evaluates multi-species thermodynamics/kinetics in a standalone module and leaves future coupling to the compressible solver as out-of-scope. This choice matches the project’s verification emphasis on the hydrodynamic discretization (MUSCL–HLLC–SSP–RK2) and on independent testing of the kinetics pipeline.

2.5 Verification and (out-of-scope) validation practices

Code credibility in CFD follows the AIAA Guide for the Verification and Validation of CFD Simulations. In that framework, verification asks “are we solving the equations right?” (code and solution verification via, e.g., method-of-manufactured solutions or canonical Riemann problems), while validation asks, “are we solving the right equations?” against experimental data of known uncertainty. Reviews by Roy and others delineate procedures for grid/time-step convergence, error estimation, and uncertainty quantification; they also caution that inter-code comparison is not validation [22].

For inviscid compressible solvers, canonical tests such as the Sod shock tube provide exact or semi-analytical references to verify shock capturing, contact resolution, and rarefaction accuracy, and are standard in establishing baseline correctness before addressing reactive coupling. Within this thesis, these practices inform the verification stance; full experimental validation of detonation physics is acknowledged as future work [23].

2.6 Positioning of this work

Summarizing the above, the solver developed here is an inviscid, finite-volume Euler code using MUSCL reconstruction with TVD limiting, HLLC fluxes with Einfeldt wave-speed estimates, and SSP-RK2 time integration; choices aligned with robust shock resolution for detonation-like flows. Chemistry and thermodynamics are implemented in a separate multi-reactor framework leveraging standard NASA polynomial data and detailed mechanisms; no reduced mechanism is used and no flow–chemistry coupling is claimed in the final integration. The background presented therefore emphasizes (i) shock-capturing methodology for compressible flow, (ii) operator-splitting strategies relevant to future coupling, (iii) accepted thermochemical data sources and detailed-kinetics tooling, and (iv) verification principles appropriate for the current, non-validated scope [15], [16], [18], [20].

3 Governing equations

This chapter establishes the mathematical model that defines the target scope of the solver and fixes notation for the remainder of the thesis. The intended flow physics are compressible, inviscid (Euler) dynamics with volumetric heat release from multi-species chemistry, formulated for an ideal-gas mixture with temperature and composition-dependent thermochemistry. Molecular viscosity and multicomponent diffusion are neglected. The implemented solver is Euler-only (no molecular heat diffusion), so heat moves by convection + pressure work through the $u(E + p)$ flux. In practice, the implemented CFD component advances the two-dimensional compressible Euler equations with variable specific heats, mixture gas constant, and ratio of specific heats derived from standard NASA polynomial thermochemistry. Chemical kinetics are treated in a separate multi-reactor module with a matching state layout, enabling future coupling but not yet interfaced with the flow solver.

3.1 State variables and assumptions

We denote conserved variables by

$$\mathbf{U} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho E \\ \rho Y_1 \\ \vdots \\ \rho Y_{N_s} \end{bmatrix}, \quad (3-1)$$

Where ρ is density, u, v are Cartesian velocity components, $E = e + \frac{1}{2}(u^2 + v^2)$ is specific total energy [24]. The $Y_k \in [0, 1]$ are species mass fractions with $\sum_k Y_k = 1$.

Primitive variables are $W = [\rho, u, v, E, Y_1, \dots, Y_{N_s}]^T$ [25]. Temperature T follows from the ideal-gas mixture equations of state $p = \rho R(T, Y)T$, with $R = \sum_k Y_k R_k$. Throughout, the standard pressure is $p^\circ = 1 \text{ bar}$ [26].

Assumptions:

- Single-temperature ideal-gas mixture; species thermochemistry from mechanism polynomials. $c_p(T, \mathbf{Y})$ and $\gamma(T, \mathbf{Y})$ are not constant.
- No body forces, no viscosity, no species diffusion no Fourier heat conduction in the implemented solver.
- Detailed chemical kinetics with N_s species and N_r reactions provided by standard mechanism.

3.2 Conservation laws for the target model

The governing PDEs in conservative form read

$$\partial_t \mathbf{U} + \partial_x \mathbf{F}(\mathbf{U}) + \partial_y \mathbf{G}(\mathbf{U}) = \mathbf{S}(\mathbf{U}) \quad (3-2)$$

with Euler fluxes [24]

$$\mathbf{F}(\mathbf{U}) = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ u(\rho E + p) \\ \rho u Y_1 \\ \vdots \\ \rho u Y_{N_s} \end{bmatrix}, \quad \mathbf{G}(\mathbf{U}) = \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ v(\rho E + p) \\ \rho v Y_1 \\ \vdots \\ \rho v Y_{N_s} \end{bmatrix}, \quad (3-3)$$

closed by the ideal-gas EOS for mixtures,

$$p = \rho R(T, \mathbf{Y})T, \quad R = \sum_k Y_k R_k, \quad H = E + \frac{p}{\rho}. \quad (3-4)$$

The chemical source terms act on energy and species only,

$$\mathbf{S}(\mathbf{U}) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \dot{q}_{chem}(T, \mathbf{Y}, p) \\ \Omega_1(T, \mathbf{Y}, p) \\ \vdots \\ \Omega_{N_s}(T, \mathbf{Y}, p) \end{bmatrix}, \quad (3-5)$$

with **mass** production rate of species k ; $\Omega_k = W_k \dot{\omega}_k$ in kgm^3s^{-1} , and the volumetric chemical heat source

$$\dot{q}_{chem} = - \sum_{k=1}^{N_s} h_k(T) W_k \dot{\omega}_k, \quad [Wm^{-3}]. \quad (3-6)$$

3.3 Detailed chemical kinetics

Let species be indexed by $N = 1, \dots, N_s$ and reactions by $r = 1, \dots, N_r$. Denote molar concentrations by $C_k = \rho Y_k / W_k$ for an ideal gas. For reaction r with stoichiometric coefficients ν'_{kr} (reactants) and ν''_{kr} (products), the net coefficient is

$$\nu_{kr} = \nu''_{kr} - \nu'_{kr}. \quad (3-7)$$

We use activities $a_k = C_k / C^\circ$. Using this convention, K_c is dimensionless and detailed balance uses K_c computed with unit activity coefficients and standard concentration $C^\circ = p^\circ / (RT)$ at the standard pressure ($p^\circ = 1 \text{ bar}$). In practice we evaluate mass-action products in concentration units and convert equilibrium constants accordingly [27].

The rate of progress is

$$\mathcal{R}_r(T, p, \mathbf{C}) = k_{f,r}(T, p) \prod_k a_k^{\nu'_{kr}} - k_{r,r}(T, p) \prod_k a_k^{\nu''_{kr}}. \quad (3-8)$$

Species molar production rates are

$$\dot{\omega}_k = \sum_{r=1}^{N_r} \nu_{kr} \mathcal{R}_r, \quad (3-9)$$

Forward rate constants use the modified Arrhenius law $k_{f,r}(T) =$

$A_r T^{\beta_r} \exp(-\frac{E_{a,r}}{RT})$ [28]. For third-body reactions we use an effective collider

concentration $[M]_r = \sum_j a_{j,r} C_j$, where $a_{j,r}$ are third-body efficiencies (default 1.0 for unspecified species) [29].

For falloff reactions (Lindemann/Troe), define the reduced pressure

$$P_r = \frac{k_{0,r}(T)[M]_r}{k_{\infty,r}(T)}, \quad k_r = \frac{k_{\infty,r}(T)P_r}{1 + P_r} F. \quad (3-10)$$

With Troe blending [30],

$$F_{cent}(T) = (1 - A)e^{-\frac{T}{T_3} +} + Ae^{-\frac{T}{T_1} + e^{-\frac{T_2}{T}}}$$

(third term includedd onlu when T_2 is specified),

$$C = -0.4 - 0.67 \log_{10} F_{cent},$$

$$N = -0.75 - 1.27 \log_{10} F_{cent}, \quad (3-11)$$

$$d = 0.14,$$

$$\log_{10} F = \frac{\log_{10} F_{cent}}{1 + \left(\frac{\log_{10} P_r + C}{N - d \log_{10} P_r + C} \right)^2}.$$

Using standard-state thermochemistry,

$$\Delta G_r^o(T) = \sum_k v_{kr} g_r^o(T),$$

(3-12)

$$K_p(T) = \exp\left(-\frac{\Delta G_r^o(T)}{RT}\right).$$

To work with concentration units (activities $a_k = \frac{C_k}{C^o}$), $K_c(T) = K_p(T) \left(\frac{p^o}{RT}\right)^{\Delta v_r}$,

$\Delta v_r = \sum_k v_{kr}$. Detailed balance gives the reverse rate $k_{r,r} = k_{f,r}/K_c$ (For elementary reactions at fixed T, p , $k_r = k_f/K_c$ follows from $K = k_f/k_r$).

These forms (ideal-gas activities, modified Arrhenius kinetics, third-body efficiencies, and Troe falloff) are matched by the implemented chemistry module.

3.4 Thermodynamic closure and derived quantities

Species standard-state properties follow the NASA polynomial parameterizations. For each species k , with coefficients $a_1, \dots, a_7$ over a temperature range, the forms of heat capacity, enthalpy, and entropy are in respective order

$$\begin{aligned}\frac{c_{p,k}^o(T)}{R} &= a_1 + a_2 T + a_3 T^2 + a_4 T^3 + a_5 T^4, \\ \frac{h_k^o(T)}{RT} &= a_1 + a_2 \frac{T}{2} + a_3 \frac{T^2}{3} + a_4 \frac{T^3}{4} + a_5 \frac{T^4}{5} + \frac{a_6}{T}, \\ \frac{s_k^o(T)}{R} &= a_1 \ln T + a_2 T + a_3 \frac{T^2}{2} + a_4 \frac{T^3}{3} + a_5 \frac{T^4}{4} + a_7.\end{aligned}\tag{3-13}$$

Two temperature ranges (low/high) are commonly used with a midpoint, or a multi-region coefficient form [31], [32].

Mixture properties follow by mass-weighted mixing rules: $h = \sum_k Y_k h_k$, $c_p = \sum_k Y_k c_{p,k}$. the internal energy is $e = h - \frac{p}{\rho}$. The mixture gas constant is $R = \sum_k Y_k R_k$. The speed of sound is $a = \sqrt{\gamma RT}$, with $\gamma = c_p/c_v$ evaluated from the mixture state.

3.5 Model scope

This thesis implements a reduced form of the target system to focus on core algorithmic components and enable later coupling. In brief:

- CFD component: 2-D compressible Euler equations for an ideal-gas mixture with temperature- and composition-dependent thermochemistry (variable c_p , γ , R). No physical viscosity or multicomponent diffusion.

- Chemistry component: A standalone multi-reactor network advanced with JAX-based ODE solvers. The state (ρ , E , Y) and mixture thermochemistry use the same data layout and naming as the CFD code to enable drop-in coupling later.
- Coupling (unimplemented): The CFD and chemistry components are not coupled in the final implementation (no source-term exchange during CFD time stepping). An interface skeleton and compatible data structures are in place, but integration was deferred due to time constraints and the complexity of robust stiff integration.
- Prototypes evaluated: A Cantera-driven network was tested but rejected for large-scale reactor nets due to runtime; a simple explicit reactor step was also tested and found inaccurate for stiff combustion even with very small time-steps. Both are summarized in chapter 4.
- Final implementations evaluated: A suite of tests was designed to evaluate the implementations produced during this project. The specifics and results are discussed in chapters 7 and 8.

This separation clarifies the contract (target model) versus the subset exercised in the results.

4 Numerical methods

This chapter describes the numerical algorithms used to discretize and advance the governing equations. The target formulation combines second-order finite-volume methods for compressible flow with operator-split updates for thermal conduction and detailed chemical kinetics. In practice, the implemented CFD solver advances only the inviscid Euler subsystem using MUSCL reconstruction, HLLE fluxes, and a strong-stability-preserving Runge–Kutta scheme. In-line chemistry coupling is retained in the model statement and outlined here for completeness but were not exercised in the final solver. Instead, a separate JAX-based multi-reactor code was developed to evaluate chemistry performance in isolation and to maintain compatibility for future coupling.

4.1 Finite-volume discretization & notation

Let $\Omega_{i,j} = [x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}] \times [y_{j-\frac{1}{2}}, y_{j+\frac{1}{2}}]$ denote a control volume with cell-average $U_{i,j}(t)$. The semi-discrete FV update in flux-difference form is

$$\frac{dU_{i,j}}{dt} = -\frac{\hat{F}_{i+\frac{1}{2}j} - \hat{F}_{i-\frac{1}{2}j}}{\Delta x} - \frac{\hat{G}_{ij+\frac{1}{2}} - \hat{G}_{ij-\frac{1}{2}}}{\Delta y} + S_{i,j} + D_{i,j}, \quad (4-1)$$

where $\hat{F}$ and $\hat{G}$ are consistent numerical fluxes obtained by solving, at each face, a one-dimensional Riemann problem in the face-normal direction. One ghost-cell layer is used on each boundary to support MUSCL reconstruction and boundary conditions.

4.2 MUSCL reconstruction and slope limiting

Piecewise-linear MUSCL reconstruction is applied to primitive variables $[\rho, u, v, p, Y_1, \dots, Y_{N_s}]$ for robustness near shocks. In 1-D notation, the slope at cell i is limited using the monotonized-central (MC) limiter

$$\sigma_i = \text{minmod}\left(\frac{U_{i+1} - U_{i-1}}{2\Delta x}, 2\frac{U_{i+1} - U_i}{\Delta x}, 2\frac{U_i - U_{i-1}}{\Delta x}\right) \quad (4-2)$$

with the multi-argument *minmod* defined component-wise by

$$\begin{aligned} \text{minmod}(a_1, \dots, a_n) &= f(x) \\ &= \begin{cases} \text{sign}(a_1) \min_k |a_k|, & \text{if } \text{sign}(a_k) = \text{sign}(a_1), \\ 0, & \text{otherwise.} \end{cases} \end{aligned} \quad (4-3)$$

left/right states at face $i + \frac{1}{2}$ are

$$U_{i+\frac{1}{2}}^- = U_i + \frac{1}{2}\Delta x \sigma_i, \quad U_{i+\frac{1}{2}}^+ = U_{i+1} - \frac{1}{2}\Delta x \sigma_{i+1}. \quad (4-4)$$

In 2-D the reconstruction is performed dimension-by-dimension. Near strong discontinuities or nonphysical updates (negative ρ, p) the method falls back locally to first order.

Species boundedness and normalization. After reconstructing Y_k at faces, enforce $Y_k \in [0,1]$ by clipping if necessary and **renormalize** to $\sum_k Y_k = 1$ (conservative rescaling of Y_k excluding any clipped species at 0 or 1). This preserves realizability and avoids spurious energy errors when converting between p, T and E .

4.3 Numerical flux: HLLE with Einfeldt speeds

At each face we evaluate an HLLE (Harten–Lax–van Leer–Einfeldt) flux in the **face-normal** direction. Let U_L, U_R denote reconstructed left/right states and let $v_n = u n_x + v n_y$ be the normal velocity for unit normal $\mathbf{n} = (n_x, n_y)$. The normal fluxes are

$$\mathbf{F}(\mathbf{U}) = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ u(\rho E + p) \\ \rho u Y_1 \\ \vdots \\ \rho u Y_{N_s} \end{bmatrix}, \quad \mathbf{G}(\mathbf{U}) = \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ v(\rho E + p) \\ \rho v Y_1 \\ \vdots \\ \rho v Y_{N_s} \end{bmatrix}. \quad (4-5)$$

Define wave-speed bounds $S_L \leq 0 \leq S_R$ using Einfeldt's estimates with Roe averages ($\tilde{\cdot}$) for robustness:

$$S_L = \min(v_{n,L} - a_L, \tilde{v}_n - \tilde{a}, 0), \quad S_R = \max(v_{n,R} + a_R, \tilde{v}_n + \tilde{a}, 0), \quad (4-6)$$

with Roe-averaged quantities

$$\begin{aligned} \tilde{u} &= \frac{\sqrt{\rho_L} u_L + \sqrt{\rho_R} u_R}{\sqrt{\rho_L} + \sqrt{\rho_R}}, & \tilde{v} &= \frac{\sqrt{\rho_L} v_L + \sqrt{\rho_R} v_R}{\sqrt{\rho_L} + \sqrt{\rho_R}}, \\ \tilde{H} &= \frac{\sqrt{\rho_L} H_L + \sqrt{\rho_R} H_R}{\sqrt{\rho_L} + \sqrt{\rho_R}}, & \tilde{v}_n &= \tilde{u} n_x + \tilde{v} n_y, \\ \tilde{a} &= \sqrt{(\gamma - 1) \left(\tilde{H} - \frac{1}{2} (\tilde{u}^2 + \tilde{v}^2) \right)}. \end{aligned} \quad (4-7)$$

The HLLE flux is then

$$\hat{\mathbf{F}}_n = \begin{cases} \mathbf{F}_{n,L}, & 0 \leq S_L, \\ \frac{S_R \mathbf{F}_{n,L} - S_L \mathbf{F}_{n,R} + S_L S_R (\mathbf{U}_R - \mathbf{U}_L)}{S_R - S_L}, & S_L \leq 0 \leq S_R, \\ \mathbf{F}_{n,R}, & 0 \geq S_R. \end{cases} \quad (4-8)$$

This choice is positivity-preserving for density under standard conditions and is robust in strong shocks. The code structure also permits an HLLC variant to sharpen contact discontinuities if desired later.

4.4 Time integration and CFL policy

For the advective operator (hyperbolic part), we use SSP [18]:

$$\mathbf{U}^{(1)} = \mathbf{U}^n + \Delta t \mathcal{L}(\mathbf{U}^n), \quad \mathbf{U}^{n+1} = \frac{1}{2} \mathbf{U}^n + \frac{1}{2} [\mathbf{U}^{(1)} + \Delta t \mathcal{L}(\mathbf{U}^{(1)})], \quad (4-9)$$

where $\mathcal{L}$ denotes the FV residual from 4.1 with $\mathbf{S}, \mathbf{D}$ omitted (handled by splitting). The global time step obeys

$$\Delta t = CFL * \min_{ij} \left(\frac{\Delta x}{|u| + a}, \frac{\Delta y}{|v| + a} \right)_{ij}, \quad (4-10)$$

using the same wave-speed estimates as the Riemann solver.

4.5 Physical admissibility and safeguards

To prevent breakdown near strong shocks, small floors are applied if needed: $\rho \leftarrow \max(\rho, \varepsilon_\rho)$, $p \leftarrow \max(p, \varepsilon_p)$. For species, enforce $Y_k \in [0, 1]$ and renormalize to $\sum_k Y_k = 1$. After any limiting or floor application in primitives, the total energy is recomputed using the EOS and mixture thermochemistry to remain consistent. These safeguards vanish with mesh refinement and do not affect smooth solutions.

4.6 Per-time-step algorithm (advective part)

- Fill ghost cells according to boundary type.
- Reconstruct limited primitive states on all faces (MUSCL/MC), including species Y_k with boundedness and normalization.
- Form left/right conservative states and compute face-normal fluxes via HLLE with Einfeldt speeds.
- Advance with SSP RK2 using the global CFL step.
- **Unimplemented Operator splitting** apply a **single explicit chemistry update** using detailed kinetics and resynchronize thermodynamic variables.

4.7 Final chemistry RHS

We formulate the cell-local reactive source term at constant volume for an ideal-gas mixture. Given the conserved state

$(\rho, \rho u, \rho v, \rho E, \rho Y_k)$, define molar concentrations $C_k = \rho Y_k / W_k$ and obtain temperature T from the mixture equation of state and thermochemistry.

For reaction r with stoichiometric coefficients ν'_{kr} (reactants) and ν''_{kr} (products), the net stoichiometric coefficient is $\nu_{kr} = \nu''_{kr} - \nu'_{kr}$. Forward rate constants follow the modified Arrhenius law $k_f(T)$,

$$k_{f,r}(T) = A_r T^{\beta_r} \exp\left(-\frac{E_{a,r}}{RT}\right). \quad (4-11)$$

pressure-dependent reactions use third-body/falloff formulas, and reverse rates follow from detailed balance with $K_c(T)$. Species production and heat release are

$$\dot{\omega}_k = \sum_r v_{kr} \mathcal{R}_r, \quad (4-12)$$

$$\dot{q}_{chem} = - \sum_k h_k^o(T) W_k \dot{\omega}_k.$$

To avoid under/overflow in products of concentrations, we evaluate in the log domain: $\ln(\prod_i C_i v_i) = \sum_i v_i \ln(\max(C_i, \varepsilon_c))$, with a small floor ε_c and then exponentiate.

After an explicit step $Y_k \leftarrow Y_k + \Delta Y_k$, enforce $Y_k \geq 0$ and renormalize $\sum_k Y_k = 1$. This prevents negative species and keeps the mixture state consistent.

For constant-volume reactors, recover temperature from internal energy: $f(T) = e(T, Y) - e_{target} = 0$, $T_{n+1} = T_n - \frac{f(T_n)}{c_v(T_n, Y)}$. Species $u_k(T)$ and $c_{v,k}(T)$ are from the NASA polynomials; mixture values are mass-weighted.

5 Software design & implementation

This chapter consolidates the software-engineering aspects of the work. It sets out the coding conventions and the optimization philosophy that encapsulates later chapters.

5.1 Platform choices

The solver was implemented in Python/NumPy to maximize portability, reproducibility, and open distribution. Python is free and open-source under an OSI-approved license, which removes licensing barriers for collaborators and makes deployments on clusters, labs, and teaching environments straightforward. In contrast, MATLAB is proprietary and requires a paid license, which constrains redistribution and automated deployments across sites [33], [34].

NumPy is the foundational array library in the scientific Python ecosystem and is used across a wide range of scientific domains; many higher-level tools (SciPy, pandas, scikit-learn, etc.) build directly on it. This widespread adoption lowers the barrier for interdisciplinary collaboration and onboarding (a key portability consideration for research code) [10].

Rather than relying on MATLAB-specific data structures, the Python port represents state with several NumPy arrays and expresses updates as vectorized operations. In NumPy, vectorized array ops and linear algebra dispatch into optimized C/Fortran routines (often linked to BLAS/LAPACK backends such as OpenBLAS or Intel MKL), which minimizes Python interpreter overhead while retaining clear, high-level code. This pattern is the standard performance idiom in scientific Python and is also what downstream libraries expect [10], [35].

To ensure repeatable setups across Windows, macOS, and Linux, we rely on isolated environments and pinned dependencies:

- **venv / virtualenv:** Python’s built-in venv creates lightweight, isolated environments (no admin rights required).
- **Conda / Miniconda:** For teams that prefer prebuilt binaries and non-Python dependencies, Conda environments provide a single-file, cross-platform specification. Miniconda offers installers for Windows, macOS, and Linux with a minimal footprint [38], [39].

In summary, Selecting Python/NumPy prioritizes open, license-free distribution; a well-understood, vectorized performance model; and mature packaging that yields one-command, cross-platform installs with reproducible environment characteristics that directly support portability for research code.

5.2 Porting strategy

The port from MATLAB to Python focused on preserving numerical behaviour through a one-to-one baseline. The first Python version mirrored the MATLAB algorithm as closely as possible; subsequent revisions changed only the data layout, not the math, to make the code runnable at scale.

5.2.1 Baseline reproduction for functional parity

A literal baseline clarifies the correspondence between implementations and provides a stable reference for regression tests. In the original MATLAB solver, the discretized-space grid is modelled as an array of structures illustrated by Figure 5.2-1 [40].

```

cell(M, N) % structure array (AoS)
├─ cell(1,1) : struct
│   ├── q : 4×1 double = [p; pu; pv; pE]
│   ├── qN : 4×1 double
│   ├── qS : 4×1 double
│   ├── qE : 4×1 double
│   ├── qW : 4×1 double
│   └─ res : 4×1 double
├─ cell(1,2) : struct
│   ├── q, qN, qS, qE, qW, res (same field names/types)
│   └─ ...
├─ ...
└─ cell(M,N) : struct
    ├── q, qN, qS, qE, qW, res
    └─ ...

```

Figure 5.2-1 Original MATLAB solver’s array of structures layout.

Field access such as *cell(i,j).q(k)* and whole-field operations reflect MATLAB’s first-class support for structure arrays and comma-separated field expansion. The MATLAB execution engine (R2015b+) compiles a broad set of language constructs, so the loop-based baseline achieves acceptable performance despite the AoS layout [41]. These language traits justify starting from MATLAB’s AoS idiom for correctness before altering the data layout.

5.2.2 Why “array-of-structures” was discarded in Python

As mentioned, MATLAB has *structure arrays* as a first-class feature, and you can pull the same field from every element of the array with a single expression. For example, writing *S.f* on a non-scalar struct returns all the *f* values (MATLAB expands it to *(S(1).f, S(2).f, ...)*, which keeps array-of-structures code compact.

In Python, trying to mirror an “array-of-structures” usually means lists of dictionaries, classes or NumPy arrays with data type defined as object. Those keep work in the Python interpreter instead of NumPy’s compiled loops, so you don’t get the speedups that come from NumPy’s model: operating on **ndarrays** (NumPy’s n-dimensional, homogeneous arrays, “one data type for all elements”) using **ufuncs** (C-implemented, element-wise functions) and

broadcasting (automatic shape expansion so loops run in C instead of Python). The result of this was that the one-to-one port was unable to be run within an acceptable execution time. The practical fix is to reorganize into a **structure-of-arrays (SoA)**, one contiguous numeric array per field.

NumPy also supports *structured (record) dtypes*, arrays whose single elements have named fields of mixed types. They're great for heterogeneous, record-like storage, but they are **not** a natural substrate for array-wide arithmetic across fields: arithmetic/bitwise operations on a structured array aren't supported; you operate **field-by-field** instead. For numerical kernels, SoA (separate homogeneous arrays) aligns better with ufuncs and broadcasting [42].

5.2.3 Baseline discrepancy on 2-D Sod (edge and corner artifacts)

To validate the one-to-one port, a two-dimensional extrapolation of Sod's shock tube in two dimensions existed in the original MATLAB code, which was ported to Python. The Python **baseline** mirrored the MATLAB algorithm and boundary conventions. Figure 5.2-2 shows the MATLAB reference at the time slice used for comparison; Figure 5.2-3 shows the Python baseline at the same time [40].

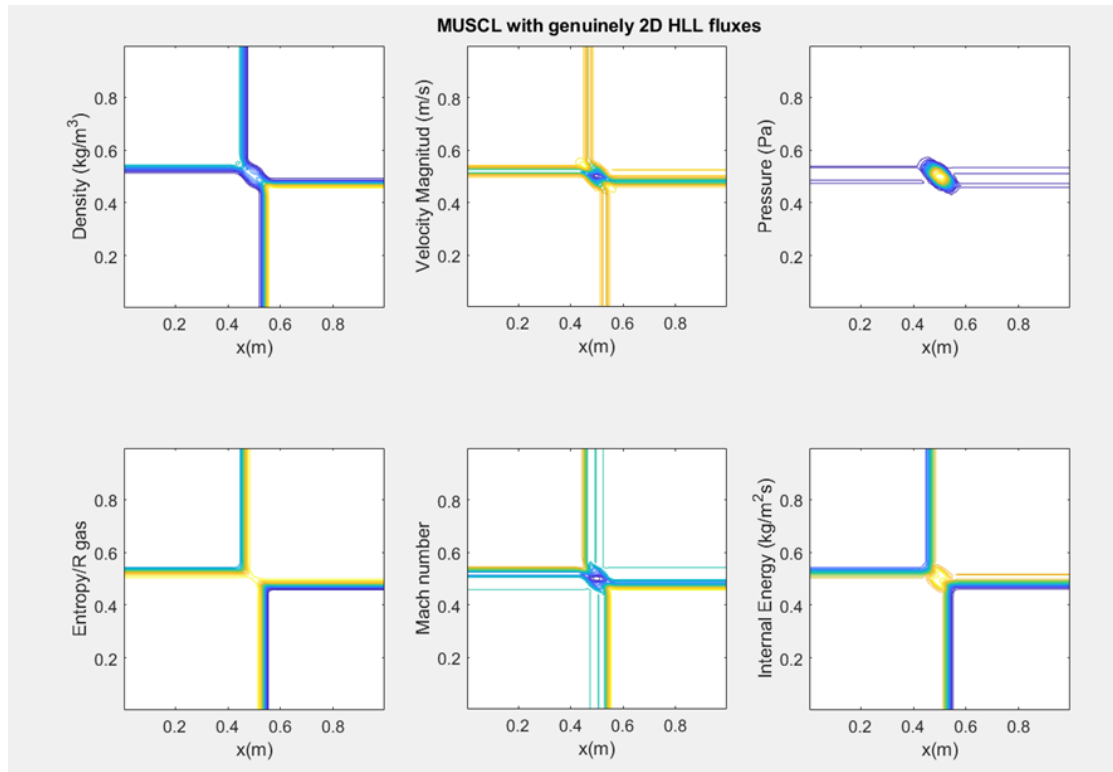


Figure 5.2-2 MATLAB reference solution for the 2-D Sod extrusion.

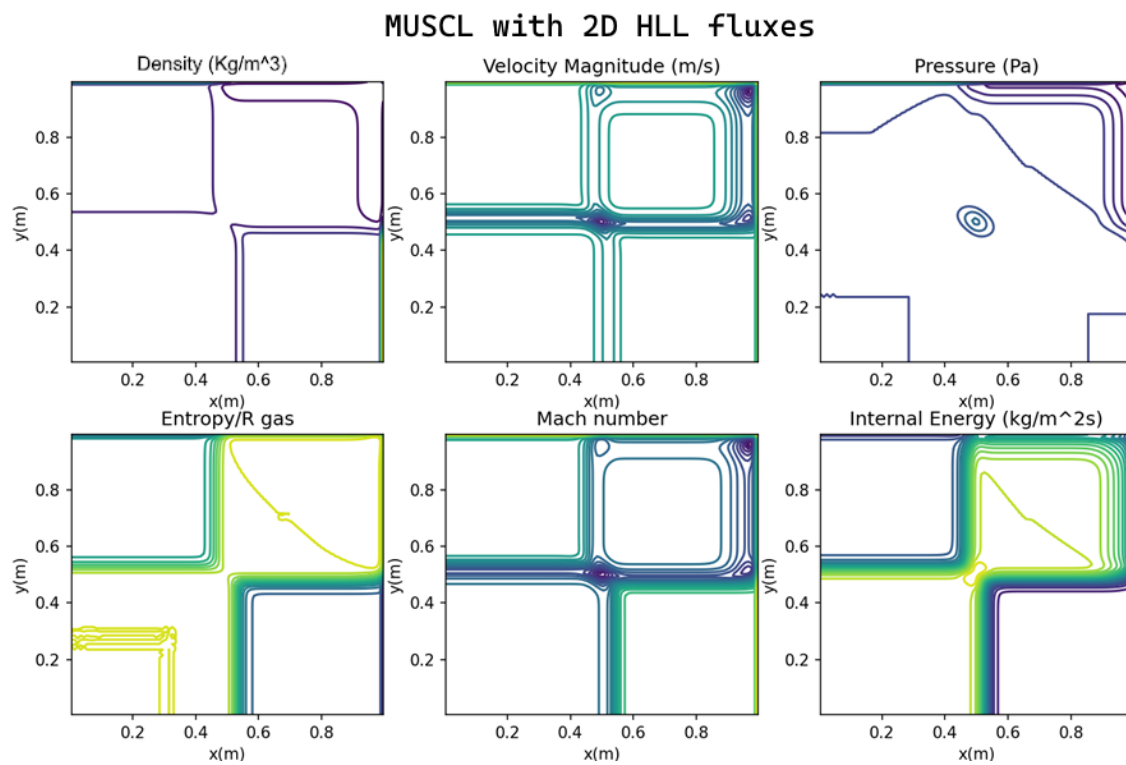


Figure 5.2-3 Python baseline solution, prior to fixes: edge streaks and corner defects are visible.

The Python field displayed large, non-physical peaks at the **rightmost** and **topmost** corners. The lower and left edges did not show the same problem, pointing to an indexing asymmetry between low and high boundaries.

Discrepancies were isolated with iteration-matched dumps, 1-D lineouts, and targeted prints of small stencils around boundaries. The investigation compared residuals and interface fluxes cell-by-cell against MATLAB until the first divergence point was located.

Concept	MATLAB (1-based, inclusive end)	Python/NumPy (0-based)
Valid array indices	$1, 2, \dots, M$	$0, 1, \dots, M$
Interior cell range	$2: M - 1$	$1: M - 2$
Loop over interior cells	<i>for</i> $i = 2: M - 1$	<i>for</i> i <i>in</i> $\text{range}(1, M - 1)$
Loop upper bound semantics	End included	End excluded

Table 5.2–1 MATLAB vs Python indexing (with one ghost layer)

The flux contribution for ghost-adjacent faces on the high side used MATLAB-style high indices verbatim in Python. From Table 5.2–1, the MATLAB code's $M-1 / N-1$ (last interior) must map to $M-2 / N-2$ in Python. Using $M-1 / N-1$ in Python instead hit the ghost indices, so fluxes were assembled with the wrong cell on the high side, producing the edge/corner spikes seen in Figure 5.2-3. In tightening the indices, a correct Python loop was accidentally changed from $\text{range}(\dots, N - 1)$ to $\text{range}(\dots, N - 2)$. From Table 5.2–1, Python's upper bound is exclusive; $\text{range}(1, N - 1)$ already covers the last interior index $N-2$. Reducing the end to $N-2$ dropped that last interior column/row, creating under-updates along the high side. Reverting the end bound restored full coverage, giving results shown in Figure 5.2-4 that were within tolerance to those obtained by the MATLAB solver, show in Figure 5.2-2 [40].

MUSCL with 2D HLL fluxes

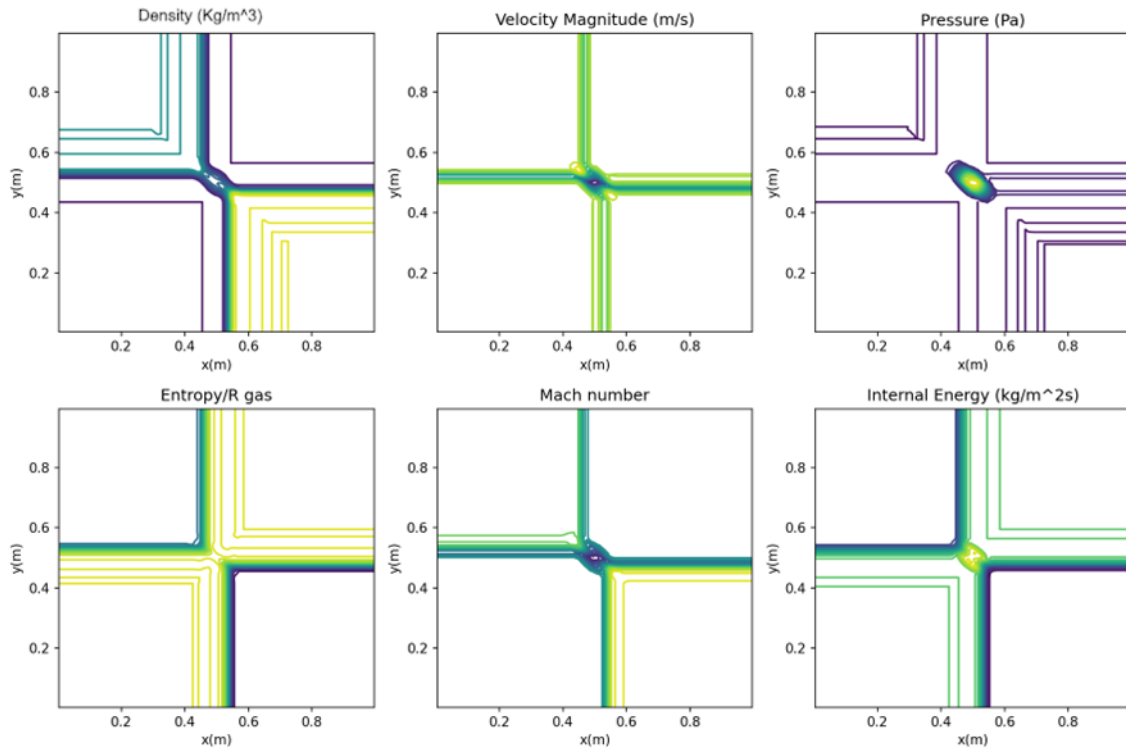


Figure 5.2-4 Python baseline solution after fixes: single-iteration density results within $e-14$ kg/m³ tolerance to those obtained by the MATLAB solver.

5.2.4 AoS to SoA transformation in the solver

After establishing functional parity, the solver's state was reorganized from an *array-of-structures* (AoS) into a *structure-of-arrays* (SoA) layout. In the SoA design, all fields of the same kind are stored together in homogeneous n-dimensional array. Concretely, transient states and residuals are allocated with shape (M, N, n_{var}) , where the last axis packs the **conservative variables** in the fixed order $[\rho, \rho u, \rho v, E]$ and, if present, species mass fractions follow. The overall structure of this implementation is shown in Figure 5.2-5 [40]. This arrangement minimizes per-cell field extraction and prepares the code for vectorized operations.

```

Top-level state (cell-centred, homogeneous ndarray):
  q : float array with shape (M, N, nvar)
      last-axis variable order:
          [ 0:  $\rho$ , 1:  $\rho u$ , 2:  $\rho v$ , 3:  $\rho E$ , 4..(3+nspecies):  $\rho Y_1 \dots \rho Y_{ns}$  ]
      - ndarray = NumPy n-dimensional, homogeneous array container.
      - nvar = 4 + nspecies (species appended after).

Intermediate face reconstructions (same shape and ordering as q):
  qE : (M, N, nvar)   # state reconstructed at each cell's East face
  qW : (M, N, nvar)   # West face
  qN : (M, N, nvar)   # North face
  qS : (M, N, nvar)   # South face

Residual accumulator (cell-centred, same layout):
  residual : (M, N, nvar)

```

Figure 5.2-5 Final Python solver's structures of arrays layout.

5.3 Data layout & vectorization

This subsection explains how the one-to-one baseline was recast into batched array code.

5.3.1 Motivation for vectorization

After establishing a one-to-one baseline, the solver moved from loop-based updates to a batched, vectorized pipeline on a structure-of-arrays (SoA) state. Here, *vectorized* means writing updates over whole $M \times N \times n_{var}$ arrays so the inner loops execute in compiled kernels rather than in the Python interpreter. NumPy's elementwise kernels and broadcasting rules run in compiled loops over contiguous, type-coherent memory regions. Writing the update in array form therefore removes per-element interpreter overhead and is one of the most significant reasons a SoA approach pays off in practice.

5.3.2 Slope reconstruction and limiters (vectorized form)

Finite-volume slope estimates are computed with array slices (selections like $q[1:-1, 1:-1, k]$). In NumPy, basic slicing returns a view, a lightweight window on the same memory, not a copy, which keeps these operations efficient in the hot path. By contrast, advanced indexing creates copies and is avoided here

[43]. In this implementation the only limiter expressed in fully vectorized form is *minmod*, shown in Figure 5.3-1 [40]; the Van Leer and Van Albada helpers remain scalar prototypes (they take scalars, branch per element, and are not invoked in the production vectorized path).

```

stacked ← stack(D1, D2, ..., Dk)           # shape: (k, ...S)
s       ← sum(sign(stacked), axis=0) / k    # average sign per location
mm      ← zeros_like(stacked[0])

mask    ← abs(s) == 1                      # true iff all signs agree
mm[mask] ← s[mask] * min(abs(stacked[:, mask]), axis=0)

return mm

```

Figure 5.3-1 Pseudo-code of the vectorized version of Harten’s *minmod* slope limiter. The mask enforces “all candidates same sign”; where signs disagree, the limiter returns zero. Where they agree, it takes the smallest magnitude and re-applies the common sign.

5.3.3 Vectorized flux evaluation

Blocks of interfaces are passed to the Riemann solver, which returns a flux array whose first two axes match the selected interface set and whose last axis is *n_var*. Residual assembly then applies each interface flux to its two neighbouring cells (add/subtract) in bulk, for both directions. This eliminates Python-level inner loops while ensuring each cell face receives its own flux; see Figure 5.3-2. Background on HLLC-family solvers is provided in the literature cited earlier [40].

<pre> for i = 1 ... M-2: for j = 1 ... N-3: qL ← qE[i, j, :] qR ← qW[i, j+1, :] Fx ← HLL1Dflux(qL, qR, axis = x) r[i, j, :] ← r[i, j, :] + Fx/dx r[i, j+1, :] ← r[i, j+1, :] - Fx/dx </pre>	<pre> qL ← qE[1:M-1, 1:N-2, :] qR ← qW[1:M-1, 2:N-1, :] Fx ← HLL1Dflux(qL, qR, axis = x) r[1:M-1, 1:N-2, :] ← r[1:M-1, 1:N-2, :] + Fx/dx r[1:M-1, 2:N-1, :] ← r[1:M-1, 2:N-1, :] - Fx/dx </pre>
---	---

Figure 5.3-2 Residual assembly in the x-direction. Left: scalar double-loop over faces; Right: vectorized update where face fluxes and residual contributions are formed and applied over slices in one step.

5.3.4 Broadcasting defect and resolution

During the transition from scalar loops to batched calls, an intermediate version of the flux routine *HLL1Dflux*, as seen in Figure 5.3-2 returned a single flux vector per row (shape (n_var) or $(1, n_var)$). When added to the residual array, NumPy’s broadcasting replicated that vector across the missing interface axes, yielding equal flux values across the entire dimensions. The issue was diagnosed via iteration-matched dumps against the MATLAB reference, which revealed identical flux along dimensions. The fix enforced batched evaluation so that the returned flux carried **both** interface axes explicitly prior to accumulation, thereby removing the inadvertent scalar path.

5.3.5 Resulting design

The final pipeline: SoA state, view-based face batching, vectorized limiters, batched HLL fluxes, and bulk residual updates, extends this section’s rationale into a concrete implementation. Array-wise operations replace inner Python loops; each face receives a distinct flux, and critical paths execute in compiled code.

5.4 Ghost-Cell Conventions and Off-by-One Errors

In a finite-volume solver, physical boundary conditions are enforced by *filling ghost cells* every sub-step, so the same face-flux kernel can run on interior and boundary faces without special cases. This is standard practice in FVM codes

(e.g. CLAWPACK/SharpClaw) and is discussed in LeVeque's text and docs [44], [45].

5.4.1 Indexing and shapes (1 ghost layer per side)

Let $nx \times ny$ be interior cells. With one ghost layer per side, cell-centred state has shape

```
q.shape = (nx + 2, ny + 2, nvar)
```

Interior cells are $q[1:-1, 1:-1, :]$.

Face-centred flux arrays (for a conservative update) are:

```
Fx.shape = (nx + 1, ny, nvar) # vertical faces at i = 0..nx
Fy.shape = (nx, ny + 1, nvar) # horizontal faces at j = 0..ny
```

so, the semi-discrete update on interior cells is

$$U'_{i,j} = -\frac{1}{\Delta x} (F_{i+1/2,j}^x - F_{i-1/2,j}^x) - \frac{1}{\Delta y} (F_{i,j+1/2}^y - F_{i,j-1/2}^y), \quad (5-1)$$

with fluxes built from left/right reconstructed states and Riemann solver.

5.4.2 Ghost-cell fill

At each Runge–Kutta sub-step we fill the four side ghost layers, then fill the corners. The mesh carries one ghost layer per side, so interior cells are indexed as $q[1:-1, 1:-1, :]$.

Outflow, shown in Figure 5.4-1 (first-order extrapolation; used for all runs in this project), copies the adjacent interior row/column to impose a zero normal gradient [40].

```

# y-min, y-max, x-min, x-max
q[:, 0, :] = q[:, 1, :]
q[:, -1, :] = q[:, -2, :]
q[0, :, :] = q[1, :, :]
q[-1, :, :] = q[-2, :, :]

```

Figure 5.4-1 Ghost cell fill pseudocode.

This approximates a non-reflecting outflow and is a standard option in CLAWPACK [46].

The reflecting wall Ghost cell fill policy copies tangential components and flips the wall-normal momentum. For a left wall (x-normal) as shown in Figure 5.4-2 [40].

```

# Left wall example (x-normal):
q[0, :, 0] = q[1, :, 0]      # rho
q[0, :, 1] = -q[1, :, 1]     # rho*u flips
q[0, :, 2] = q[1, :, 2]      # rho*v copies
q[0, :, 3] = q[1, :, 3]      # rho*E copies
q[0, :, 4:] = q[1, :, 4:]    # species copy

```

Figure 5.4-2 Reflecting wall ghost cell fill (available for code verification).

Apply the analogous transformation on the other sides by flipping the momentum component aligned with the local outward normal [44].

Uniform fields remain uniform to machine precision after a ghost fill; wall-normal mass flux at reflecting boundaries is (numerically) zero; only interior slices $q[1:-1, 1:-1, :]$ are advanced by the update.

5.5 Debugging methodology

The debugging strategy was designed to isolate numerical faults quickly and to make fixes reproducible. It combines deterministic iteration-matching against a trusted reference, local numerical probes (stencil dumps), one-step unit tests on residuals and conservation, and lightweight visual diagnostics.

A MATLAB run of the original implementation served as a reference. For a fixed initial state, CFL, and boundary treatment, the Python port and the MATLAB

solver were made to dump their results at each timestep. These arrays were compared (e.g., reconstructed face states, Riemann fluxes, cell-centred residuals). Any divergence then narrowed the search to a single kernel (ghost fill -> reconstruction -> Riemann -> update). This process exposed a MATLAB->Python off-by-one translation at the outer indices in ghost-cell flux contributions and an incorrect loop bound arising from Python's half-open ranges, both caught within the same framework.

To avoid "plot and guess," selective printing of small 3×3 or 5×5 stencils around suspect cells was used to inspect raw numbers at full precision (density, momenta, energy, reconstructed left/right states, and resulting fluxes). These probes were gated by simple predicates (e.g., "largest residual," "positivity clamp triggered") and were essential for tracing spurious corner peaks to an indexing mistake in the ghost-cell treatment.

Interactive stepping (Python debugger) placed breakpoints at the BC fill, MUSCL reconstruction, Riemann call, and update. Watches tracked index windows and array shapes to surface silent broadcasting mistakes. This is how a row-wise flux bug (a single flux reused across a row due to a non-vectorized flux kernel) was diagnosed after several days of investigation; once fixed, it aligned the Python port with the verified implementation up to floating-point differences.

Where numbers converged but intuition still flagged issues, quick 1-D cuts (e.g., centreline pressure/density) were plotted to reveal repeating artifacts characteristic of broadcasting/loop-bound errors, and to confirm that fixes removed corner anomalies.

After fixes and full vectorization of the flux pipeline, the Python solver matches the reference within expected floating-point tolerances and runs with comparable wall-clock times on like-for-like cases; subsequent discrepancies are now attributed to precision/implementation choices rather than logic errors.

5.6 Performance Engineering

The aim here was practical: record indicative timings that helped steer development, not to present a full benchmarking study. Figures below are end-to-end wall times gathered during implementation and used only as baselines for usability and prioritisation. Hardware, BLAS backend, grid size, and case setup all materially affect results, so numbers should be read qualitatively.

Pre-vectorization on a toy “circular over-pressure” blast case, advancing 0.3 s of simulated time required ~30 s wall time; extending to 1.0 s required ~120 s. These figures motivated a shift away from scalar/looped code toward a fully vectorized pipeline.

After rewriting the flux pipeline in vectorized form, wall time decreased substantially relative to the pre-vectorized code, with overall runtimes significantly better than the MATLAB reference on mirrored cases. Because outcomes vary with mesh size, this is reported as a qualitative improvement rather than a fixed speedup factor.

These timings are case-level snapshots, not normalized per-cell or per-step metrics, and they exclude systematic sweeps across hardware or problem sizes. See chapter 8 for quantitative results.

5.7 Orchestration & Object-Oriented Design

This section records the *software architecture* used to implement the Euler solver and to prototype chemistry hooks. It justifies an object-oriented wrapper and documents the public interface; it does not claim coupled chemistry results.

```
CFDGrid
├─ Geometry:  Lx, Ly, nx, ny, dx, dy
├─ Species:   species_names, nspecies, species_index
├─ State:      $q \in \mathbb{R}^{ny \times nx \times (4 + nspecies)}$  # with ghost layers
├─ Chemistry: cantera_reactors[j][i] (interior only), ReactorNet, mechanism
├─ Public API (selected):
│   • initialize_euler(rho, rho_u, rho_v, rho_E)          -> set interior q
│   • create_consistent_initial_state(P, Y, T or e)      -> thermodynamic init
│   • add_species(name), initialize_chemistry(Y, T, mech) -> reactors +  $\rho Y_k$ 
│   • muscl_euler_res2d_v0 / _v1(...)                   -> build residuals
│   • apply_reflecting_bc(side, ng=1)                   -> mirror & flip normal
│   • advance_chemistry(dt)                             -> step ReactorNet, write back
│   ...
└─ Usage: ghost-fill -> residual -> SSP-RK2 update
```

Figure 5.7-1 Schematic of the implemented CFDGrid object.

In justifying the object-oriented approach, the following considerations were taken.

- **Modularity.** Encapsulating state and numerics allows swapping reconstruction/flux routines or chemistry backends without changing calling code.
- **Single source of truth.** The class owns the conserved state including ghosts, grid metrics, and species metadata, so indexing conventions are centralized and consistent.
- **Safety & invariants.** Boundary fills, realizability checks, and reactor housekeeping can live next to the data they protect, reducing off-by-one and broadcasting errors.
- **Testability.** Methods become natural seams for tests.

- **Future extensibility.** Chemistry/energy transport modules can plug into a stable interface later without reworking the solver loop.

Object-oriented design is used here to keep the solver's moving parts modular and testable. By bundling data (mesh, state arrays, species) with the routines that operate on them (ghost fills, reconstruction, Riemann solver, optional chemistry hooks) into a single class, structurally shown in Figure 5.7-1, we can swap implementations and extend the code without rewriting the whole pipeline [40]. This approach is widely used in mature scientific PDE libraries such as OpenFOAM and others. In scientific software practice, this kind of modularization is recommended because it improves maintainability, re-use, and comprehensibility of code bases [47], [48], [49].

6 Thermochemistry

This chapter documents the design process of the thermochemistry component that accompanies the Euler solver. It summarizes modelling choices (reduced chemistry and Arrhenius rates), the intended operator-splitting orchestration, early prototyping and the final implementation. The emphasis is architectural and methodological; quantitative performance and coupling results are deferred to chapters 7 and 8. Deeper insights into implementation can be found in 10.4 Appendix B.

6.1 Scope note

Chemistry was not coupled to the production CFD results. A chemistry prototype was implemented and tested in isolation but not integrated with the transport step due to time constraints.

6.2 Prototype A: Cantera ReactorNet

As the first approach, we coupled each CFD step to a Cantera ReactorNet: batched cell (or subnetwork) states (ρ^n, u^n, E^n, Y^n) were mapped to reactors and integrating the assembled network as a single ODE system via SUNDIALS (CVODES/IDAS) under Lie splitting [50].

This prototype was conceptually clean and allowed rapid prototyping. However, it did not scale; Cantera does not use multiple threads internally, and safe parallel use requires separate copies of Cantera objects per thread/process; practical parallelism is therefore user-managed (e.g., one ReactorNet per worker). At realistic grid sizes, wall-clock time per CFD step became prohibitive [51], [52].

Because of these limitations, the initial prototype was ultimately discarded. It nevertheless clarified the requirements for the chemistry module: an optimized implementation compatible with explicit CFD stepping and scalable to many cells. These insights directly motivated a simpler single-step approach.

6.3 Prototype B: “rates-only” explicit step

To reduce overhead, we bypassed reactor objects and batched calls to Cantera for source terms Ω_k and $\dot{q}_{\text{chem}}$ then applied a single explicit Euler update.

$$\begin{aligned}\Delta Y_k &= \frac{\Omega_k(T^n, \mathbf{Y}^n, p^n)}{\rho^n} \Delta t, & \Delta E &= \frac{\dot{q}_{\text{chem}}(T^n, \mathbf{Y}^n, p^n)}{\rho^n} \Delta t, \\ Y_k^{n+1} &= Y_k^n + \Delta Y_k, & E^{n+1} &= E^n + \Delta E,\end{aligned}\tag{6-1}$$

with realizability limiting

$$\alpha = \min\left(1, \min_{k: \Omega_k > 0} \frac{\rho^n Y_k^n}{\Delta t \Omega_k}\right), \quad \Delta Y_k \leftarrow \alpha \Delta Y_k, \Delta E \leftarrow \alpha \Delta E.\tag{6-2}$$

Despite having a sequential step in determining Ω_k and $\dot{q}_{\text{chem}}$ this approach was significantly faster than the initial one, making quick time execution of large grids feasible. Despite this, for stiff combustion chemistry, a single explicit source step remained first-order and inaccurate without sub-cycling or an implicit solve.

Both prototypes clarified that the chemistry component needed a lightweight batched RHS compatible with explicit CFD stepping and parallel hardware, motivating the standalone JAX implementation. This aligns with broader practice: specialized toolchains such as TChem and pyJac target performance-portable evaluation of kinetics/Jacobians on CPUs/GPUs, underscoring why general-purpose Cantera is not ideal for massive reactor nets [21], [53].

6.4 Software implementation of the chemistry RHS

The production chemistry code is a standalone multi-reactor right-hand side (RHS) used for verification and performance exploration. It shares data structures and thermochemistry with the CFD code so that coupling can be added later. In practice, the module evaluates the method in 4.7 on many cell-local reactors at once, keeping array shapes and species ordering consistent with the flow solver so the hand-off is straightforward when coupling is enabled. Pseudo-code landmarks of the implementation can be found in 10.4 Appendix B.

The RHS is evaluated over many reactors using array programming and batched tensor contractions. This avoids Python-level loops, reduces orchestration overhead relative to per-reactor calls, and is compatible with later JIT compilation.

```
RHS_grid(t, Y_all):  
    # Y_all has shape (Ncells, S+1)  
    return vmap(RHS_cell over cells)(t, Y_all, PARAMS)
```

Figure 6.4-1 Pseudocode of Batched evaluation across a grid: Turns the cell RHS into a batched RHS for N reactors.

The module is organized around two entry points: a per-cell RHS *rhs_cell*(*Y*, *e*, *rho*) (Figure 6.4-2 [40]) that computes time derivatives of *Y* and *e* using the thermochemistry defined in 4.7, and a grid integrator *integrate_grid*(*Y0*, *e0*, *rho*, *t_span*, ...) (Figure 6.4-1 [40]) that batches cells for time integration. Realizability safeguards (species clipping and renormalization; energy positivity) and the temperature, internal-energy recovery ($T \leftrightarrow e$) are enforced internally so callers receive consistent states.

```

RHS_cell(t, state, PARAMS):
  Y ← clip(state[1:S], 0, 1) ; e ← state[S+1]
  Y ← renormalize(Y)

  # T from e for constant-volume formulation
  T ← T_from_e(e, Y)

  # (optional) clamp T to a physically safe band
  # clips are diagnostic guards and are not intended to change the physics; alarms are raised
  T ← clip(T, 300, 3500)

  # precompute Kc and then rates/production
  Kc ← precompute_Kc(T)
  (wdot, _, _) ← ROP_&wdot(T, Y, p, MECHANISM, Kc)

  # mass-fraction time derivative (mass basis)
  dYdt = (wdot * W) / ρ

  # internal-energy time derivative (chemistry heat release)
  # u(T) from NASA-7; constant-volume ⇒ use u, not h
  u_k ← species_internal_energy(T)
  dedt = - dot(u_k, wdot) / ρ
  dedt ← clip(dedt, min,max) # numeric safety

  # enforce  $\sum Y = 1$  via orthogonal projection
  dYdt ← dYdt - ( $\sum dYdt / \sum Y$ ) * Y

  return concat(dYdt, dedt)

```

Figure 6.4-2 Pseudocode of Cell-local RHS: The per-cell derivative $[\dot{Y}, \dot{e}]$, enforcing realizability and mass conservation.

Data structures (conserved arrays, species layout, mixture properties) mirror those in the CFD code to minimize conversions and enable a drop-in operator-split coupling. The current implementation favours contiguous memory views and a structure-of-arrays layout in performance-critical paths.

The batched RHS is wrapped in an *ODETerm* and advanced with Difffrax. For non-stiff problems, *Tsit5* is a good general-purpose explicit RK; *Dopri5* is a strong general default (and MATLAB’s ode45 analogue); for stiffness, an implicit ESDIRK such as *Kvaerno3* is available. We empirically default to a *Kvaerno3*

-style configuration with tuned tolerances and provide a *use_stiff/solver_type* switch for experiments [54]. Implementation shown on Figure 6.4-3 [40], [55].

```
IntegrateGrid(y0, t0, t1; use_stiff, solver_type, rtol, atol):
    term = ODETerm(RHS_grid)
    solver = choose:
        if solver_type given: {Tsit5, Dopri5, Kvaerno3}
        else if use_stiff:    Kvaerno3 # practical default in code
        else:                 Tsit5
    steps = PIDController(rtol, atol, tuned coefficients for stiffness)
    save = SaveAt(t1=True)      # last time only for throughput
    sol = diffeqsolve(term, solver, t0, t1, dt0=1e-8, y0, args=(MECH, p),
                      stepsize_controller=steps, max_steps=... , throw=False)
    return sol
```

Figure 6.4-3 Pseudocode of Time integration with DiffraX:

6.5 Integration plan with CFD

Transport and reactions are decoupled with Lie splitting at present. The workflow is:

- Advance the finite-volume transport on (ρ^n, u^n, E^n, Y^n) .
- Convert to primitive variables and temperature.
- Apply a reaction update to species mass fractions and internal energy.
- Resynchronize conserved variables. This modular structure allows the reaction step to be swapped between backends (Jax model or Cantera).

7 Verification

7.1 CFD tests

This section specifies the verification tests applied to the finite-volume Euler solver. In Table 7.1–1 each CFD test’s intent, setup and metrics are summarized.

Test (entry point)	What it demonstrates	Minimal setup (as scripted)	Primary metrics
Sod shock tube (<code>run_2d_sod_with_grid(...)</code>)	Solution verification vs. exact Riemann solution; reconstruction + flux integrity; optional MATLAB parity.	$n_x \approx 200$, $n_y = 1$; Sod IC; MUSCL–MC + HLLC; SSP-RK2; outflow ghosts; $t_{\text{end}} \approx 0.15$.	Divergence between exact solution conservation variables; optional Δ vs MATLAB.
Circular blast (non-vectorized) (<code>run_blast_test(...)</code>)	Performance baseline for residual kernel v0; qualitative 2-D flow sanity.	Typical: $n_x = n_y \approx 100$; central over-pressure; MUSCL–MC + HLLC; SSP-RK2; outflow ghosts; kernel v0.	Wall time per run.
Circular blast (vectorized) (<code>run_blast_test_vectorized(..)</code>)	Performance of vectorized residual kernel v1 with identical initial conditions.	Same as Test 2 but kernel v1.	Wall time per run.
Vectorization benchmark harness (<code>benchmark_vectorization()</code>)	Scaling & cross-code timing sweep; collects Tests 2–3 across sizes; optionally overlays MATLAB timings.	Size sweep over $N \times N$; calls Tests 2 and 3; optional MATLAB CSVs; produces plot.	Runtime vs N : speedup vs N compared to MATLAB reference.

Table 7.1–1 CFD verification tests with intent, setup and metrics.

The focus of this subchapter is on what each test demonstrates and what information it provides about solver correctness, boundary handling, and performance posture. The corresponding plots, figures, numbers and analysis appear in chapter 8.

7.1.1 Sod shock tube

The Sod problem serves as a compact, standard check on a 1-D Riemann problem [22], [23], [24], [56]. The case is run in a thin 2-D strip so that the same reconstruction/flux machinery as the 2-D solver is exercised. Assessment is carried out by comparing the primary flow variables—density (ρ), velocity (u),

and pressure (p)—at a common final time across three sources: this solver, an exact solution, and a MATLAB implementation.

The post-processing reports, for each variable and each pairing (Python vs exact; Python vs MATLAB):

- The **average absolute difference** across the domain.
- The **maximum absolute difference**.
- The corresponding **relative (percent) differences** with respect to the chosen reference curve.

Overlays of the three curves are included for orientation, but the numerical summaries are the basis of the comparison. These scalars give a direct, implementation-agnostic indication of agreement without invoking formal error norms.

7.1.2 Non-vectorized circular blast

The circular blast configuration consists of a central over-pressure that expands outward into a uniform medium. It serves two purposes. First, it provides a simple visual check that the solver maintains radial symmetry and produces physically consistent wave fronts in two dimensions. Second, it establishes a performance baseline for the un-vectorized residual implementation.

The main information extracted from this test is the wall-clock time required for a fixed number of steps, alongside qualitative inspection of the density and pressure fields. Any strong departure from radial symmetry, or the appearance of irregular artefacts, would indicate indexing or boundary handling errors. The measured runtime provides the reference against which later optimizations are compared.

7.1.3 Vectorized circular blast

The vectorized circular blast test mirrors the non-vectorized version in its physics and initial conditions: a central over-pressurized region that expands into a quiescent ambient field. Its purpose is to confirm that the same physical

solution is obtained via the optimized, vectorized residual code path, while providing a comparable performance expectation.

The same qualitative checks, symmetry of pressure and density rings, absence of visible artefacts, remain in place. Additionally, the runtime from this test directly enables calculation of speedup factors when compared to the non-vectorized baseline. Consistency in solution appearance confirms correctness; reduced execution time demonstrates the efficiency gain achieved through vectorization.

7.1.4 Vectorization benchmark harness

The vectorization benchmark harness evaluates how runtime performance scales with problem size under both non-vectorized and vectorized residual implementations. The test sweeps a range of grid dimensions and records the wall time required for each configuration on both code paths. Optionally, externally provided MATLAB timings are overlaid for cross-comparison when available. This setup supports two primary insights:

- Whether execution time scales predictably with grid size
- The magnitude of speed-up achieved through vectorization.

Monitoring the resulting curves also guards against anomalous performance behaviour.

7.2 Chemistry tests

This section specifies the verification tests aimed at the chemistry module independently of CFD. In the Table 7.2–1 each Chemistry test's intent setup and metrics are reported.

Test (entry point)	What it demonstrates	Minimal setup (as scripted)	Primary metrics
Thermodynamic conversion parity (<code>debug_thermo_conversion</code>)	Validates NASA-7 thermodynamics (cp, h, s) and $e \leftrightarrow T$ conversions against Cantera at sampled temperatures.	Build mechanism (e.g., <code>h2o2.yaml</code>); choose temperature grid; evaluate JAX and Cantera thermodynamic properties.	Absolute/relative differences for cp, h, s, $e(T)$.
Instantaneous kinetics parity (<code>debug_rates_once</code>)	Validates species production rates and chemistry heat-release at a single state; isolates rate evaluation from integration.	Build mechanism; choose one (T, rho, Y); compute $K_c(T)$; evaluate JAX $\dot{w}$ / $\dot{q}$ and Cantera at the same state.	Absolute/relative differences of $\dot{\omega}_k$; agreement of $\dot{q}_{chem}$.
Comprehensive Chemical Kinetics Validation Matrix (<code>demo_debug</code>)	Validates instantaneous rates and chemistry heat-release across a 3×6 matrix of states (lean/stoich/rich $H_2/O_2 \times 600\text{--}2000\text{ K}$); produces heatmaps for quick inspection.	Build mechanism (e.g., <code>h2o2.yaml</code>); define three mixtures (e.g., 3%, 10%, 15% H_2); temperatures {600, 800, 1000, 1200, 1500, 2000} K; evaluate JAX vs Cantera per state.	Max/avg relative differences (%) per state for $\dot{\omega}_k$ and $\dot{q}_{chem}$; summary table and heatmap.
JAX vs Cantera scaling benchmark (<code>benchmark_cantera_vs_jax</code>)	Throughput and scaling across grid sizes; quantifies speed-up vs per-cell Cantera reactors.	Size sweep over $N \times N$; identical ICs per cell; identical horizon; fixed solver choice.	Runtime vs N; speed-up (Cantera / JAX).
Divergence vs initial condition and runtime (<code>compare_initial_vs_runtime_divergence</code>)	Maps divergence (%) versus {T or p} and integration time to identify regimes where JAX and Cantera separate; highlights stiff regions and safe steps.	Grids of {T or p} $\times$ list of t_{end} ; single-cell comparisons; common mechanism and EOS.	Divergence surfaces (%).

Table 7.2–1 Chemistry verification tests with intent, setup and metrics.

Much like the previous one, the focus of this subchapter is to spell out each test’s setup, the information it provides and what can be inferred from it. The corresponding plots, figures, numbers and analysis appear in chapter 8

7.2.1 Thermodynamic conversion parity

The bidirectional conversion between temperature and energy represents a critical numerical component of constant-volume reactive flow simulations. This test systematically evaluates the numerical accuracy and robustness of the

JAX-based NASA-7 thermodynamic implementation by examining temperature-to-energy ($T \rightarrow e$) and energy-to-temperature ($e \rightarrow T$) conversions across a wide range of temperatures (300K-2500K).

The test methodology involves a two-stage process that confirms thermodynamic consistency:

1. Forward conversion: calculating internal energy from temperature using both JAX and Cantera
2. Inverse recovery: reconstructing temperature from the calculated energy using the Newton iteration scheme

For a representative hydrogen-oxygen mixture, the assessment reports:

- Absolute energy differences between JAX and Cantera at identical temperatures.
- Roundtrip temperature recovery error ($T \rightarrow e \rightarrow T$).
- Energy-to-temperature conversion accuracy using Cantera's energy values as input.
- Temperature resolution across the entire operating range (300K-2500K).

7.2.2 Instantaneous kinetics parity

This test evaluates the instantaneous chemical source terms computed by the JAX and Cantera implementations for a specific thermodynamic state, representing a snapshot of the more comprehensive validation performed by the Comprehensive Chemical Kinetics Validation Matrix test outlined in the following section. While this isolated test provides valuable insights into numerical accuracy at a specific condition, it is most meaningful when viewed as part of the broader validation matrix spanning diverse temperatures and compositions.

For a single initial state, the test compares instantaneous species production rates between the JAX implementation and Cantera when evaluated at identical states. The assessment reports:

- The absolute and relative differences for each species production rate.

- The mean and maximum relative differences across all reactions.
- The relative difference in the overall energy source term.
- The normalised root-mean-square difference between the two implementations.

7.2.3 Comprehensive Chemical Kinetics Validation Matrix

The comprehensive chemical kinetics validation framework establishes definitive verification of the JAX implementation across diverse combustion-relevant conditions. This test systematically evaluates instantaneous reaction rates across a carefully constructed matrix of thermodynamic states spanning multiple temperatures (600K-2000K) and mixture compositions (lean, stoichiometric, and rich H_2/O_2 mixtures) to encompass the full operating envelope of hydrogen combustion.

The test methodology implements a structured validation approach:

1. Construction of representative mixtures with appropriate radical seeding to avoid numerical instabilities
2. Evaluation of identical thermodynamic states in both JAX and Cantera implementations
3. Compilation of comprehensive statistics across all test conditions
4. Summary analysis of maximum discrepancies and energy equation consistency

For each thermodynamic state, the assessment reports:

- Absolute and relative differences in species production rates for all chemical species.
- Maximum relative difference across all reactions for each condition.
- Relative difference in the overall energy source term.
- Statistical summaries organized by mixture composition and temperature.

7.2.4 JAX vs Cantera scaling benchmark

The computational performance benchmark provides quantitative assessment of the JAX-based chemistry implementation's acceleration capabilities compared to the reference Cantera implementation. This test systematically evaluates the computational efficiency of both implementations across a range of problem sizes (2×2 to 14×14 grids) for a constant-volume hydrogen-oxygen combustion scenario at 1500K initial temperature.

The benchmark methodology implements a structured comparative approach:

1. Construction of identical initial conditions for both implementations to ensure fair comparison.
2. Time-accurate integration of chemical kinetics for a fixed physical time period (10^{-5} seconds).
3. Measurement of wall-clock execution time for both implementations at each problem size.

Evaluation of scaling behaviour as problem size increases.

- For each tested problem size, the assessment reports:
- Absolute execution time for both JAX and Cantera implementations
- Computational speedup ratio (Cantera time / JAX time)
- Final temperature prediction from both solvers
- Temperature difference between implementations
- Scaling efficiency relative to ideal linear scaling

7.2.5 Divergence vs initial condition and runtime

The integration stability analysis systematically evaluates the numerical robustness of the JAX chemistry implementation across diverse thermodynamic conditions and integration timescales. This test quantifies the divergence between JAX and Cantera implementations as a function of three key variables: initial temperature, initial pressure both across integration time span, providing a comprehensive map of solution instability regions.

The test methodology implements a structured parametric approach:

1. Construction of two one-dimensional parameter spaces for initial conditions:
 - a. Temperature variation (800K-2200K) at constant pressure (1 atm).
 - b. Pressure variation (0.5 to 20 atm) at constant temperature (1500K).
2. Execution of chemical kinetics simulations across multiple integration timescales (10^{-5} to 10^{-3} s).
3. Measurement of temperature prediction differences using Cantera as the reference solution
4. Visualization of divergence patterns using three-dimensional surface plots

For each thermodynamic state and integration time, the assessment reports:

- Absolute temperature values from both JAX and Cantera implementations
- Relative percentage difference in final temperature predictions
- Patterns of divergence across the parameter space
- Identification of critical regions where numerical challenges arise

8 Results

8.1 CFD test results

This section reports the outcomes of the CFD tests introduced in the previous chapter. Each subsection gives a brief orientation to the test, then presents what is reported (plots/tables) and how to read them. Method intent and set-up details are not repeated here; see Chapter 7 for those.

8.1.1 Sod shock tube results

Figure 8.1-1 and Figure 8.1-2 compare the Python solver, the exact solution, and the MATLAB implementation for the primary variables (ρ, u, p, E). Table 8.1-1 (Python vs MATLAB) and Table 8.1-2 (Python vs Exact) report the average absolute difference, maximum absolute difference, and the corresponding relative (percent) differences for each variable [40].

Python vs MATLAB				
Variable	Average Absolute Delta	Maximum Absolute Delta	Average Relative Delta (%)	Maximum Relative Delta (%)
Density	0.000677	0.019988	0.235041	15.15165
Velocity	0.002274	0.187812	2.119522	306.4409
Pressure	0.000715	0.02773	0.285346	25.63727
Energy	0.003914	0.215792	0.147449	10.51758

Table 8.1-1 Sod's shock tube results; MATLAB quantitative divergence.

Against MATLAB (Table 8.1-1), average differences are small across all fields; on the order of 10^{-3} to 10^{-2} : $\rho = 0.000677$, $u = 0.002274$, $p = 0.000715$, $E = 0.003914$.

Average relative differences are sub-percent for density (0.24%), pressure (0.29%), and energy (0.15%), and around 2.12% for velocity. The maxima are localized values near discontinuities ($\rho_{\text{max}} = 0.01999$, $u_{\text{max}} = 0.1878$, $p_{\text{max}} = 0.02773$, $E_{\text{max}} = 0.2158$). The extremely large maximum relative percentages for velocity (306%) arise where the reference value is close to zero (e.g., in quiescent regions), which inflates percentage metrics despite small absolute differences. The overlays in Figure 8.1-1 and Figure 8.1-2 are consistent with these numbers: Python and MATLAB tracks are visually coincident across most of the domain.

SSP-RK2 TVD-MUSCL Euler Equations (2D Sod Tube)

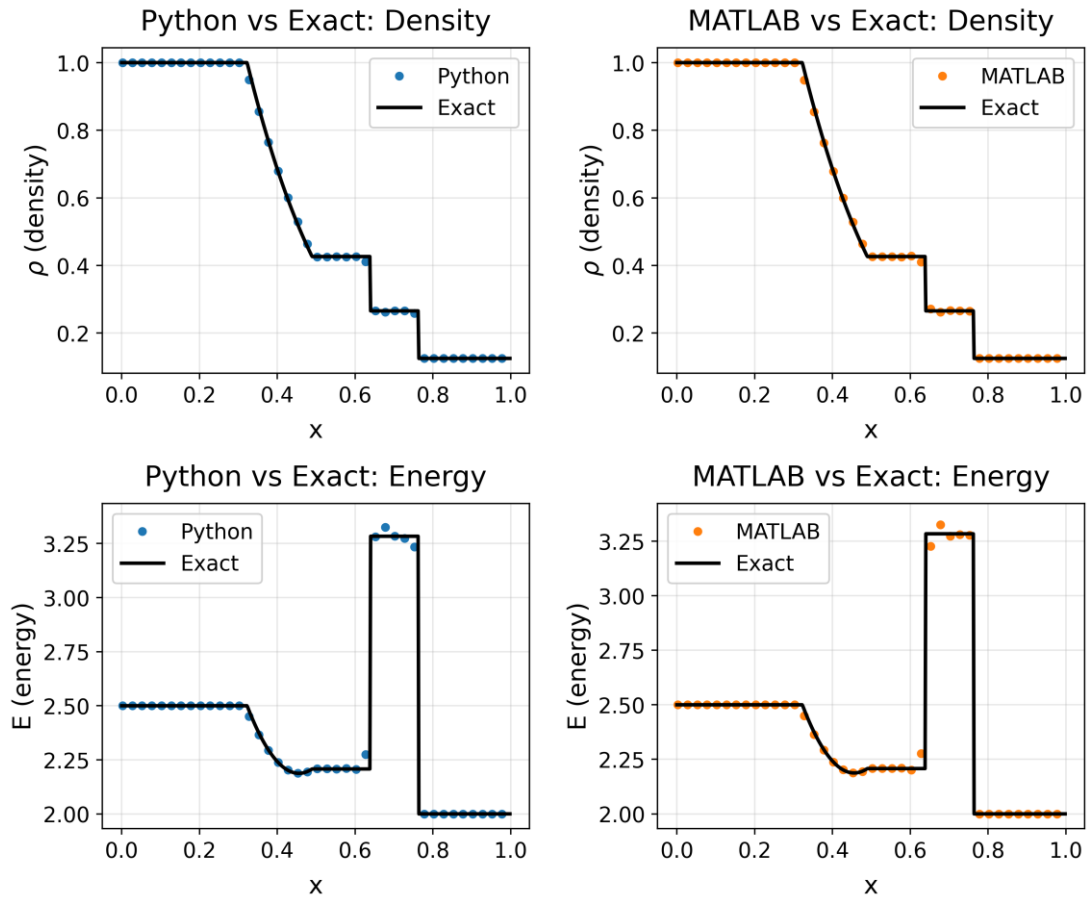


Figure 8.1-1 Sod's shock tube results; Density and Energy visual comparisons.

Python vs Exact				
Variable	Average Absolute Delta	Maximum Absolute Delta	Average Relative Delta (%)	Maximum Relative Delta (%)
Density	0.003656	0.073973	1.129424	59.17866
Velocity	0.008365	0.574077	10.46114	1748.774
Pressure	0.003121	0.102794	1.166031	102.7937
Energy	0.017201	0.712784	0.678207	35.63919

Table 8.1–2 Sod's shock tube results; Exact solution quantitative divergence.

Against the exact solution (Table 8.1–2), the average absolute differences remain modest: $\rho = 0.003656$, $u = 0.008365$, $p = 0.003121$, $E = 0.017201$. The corresponding average relative differences are $\approx 1.13\%$ (ρ), 10.46% (u), 1.17% (p), and 0.68% (E). As before, the very large maximum relative percentages, e.g., $1,748.8\%$ for velocity and 102.8% for pressure, are a consequence of

dividing by values that are near zero in parts of the reference curve; the maximum absolute differences (0.5741 in u , 0.1028 in p , 0.0740 in p , 0.7128 in E) indicate that the largest pointwise deviations are confined to narrow regions around discontinuities, which dominate the max metrics but not the averages. The overlays in Figure 8.1-1 and Figure 8.1-2 reflect this: the curves coincide over most of the domain and separate primarily near sharp transitions.

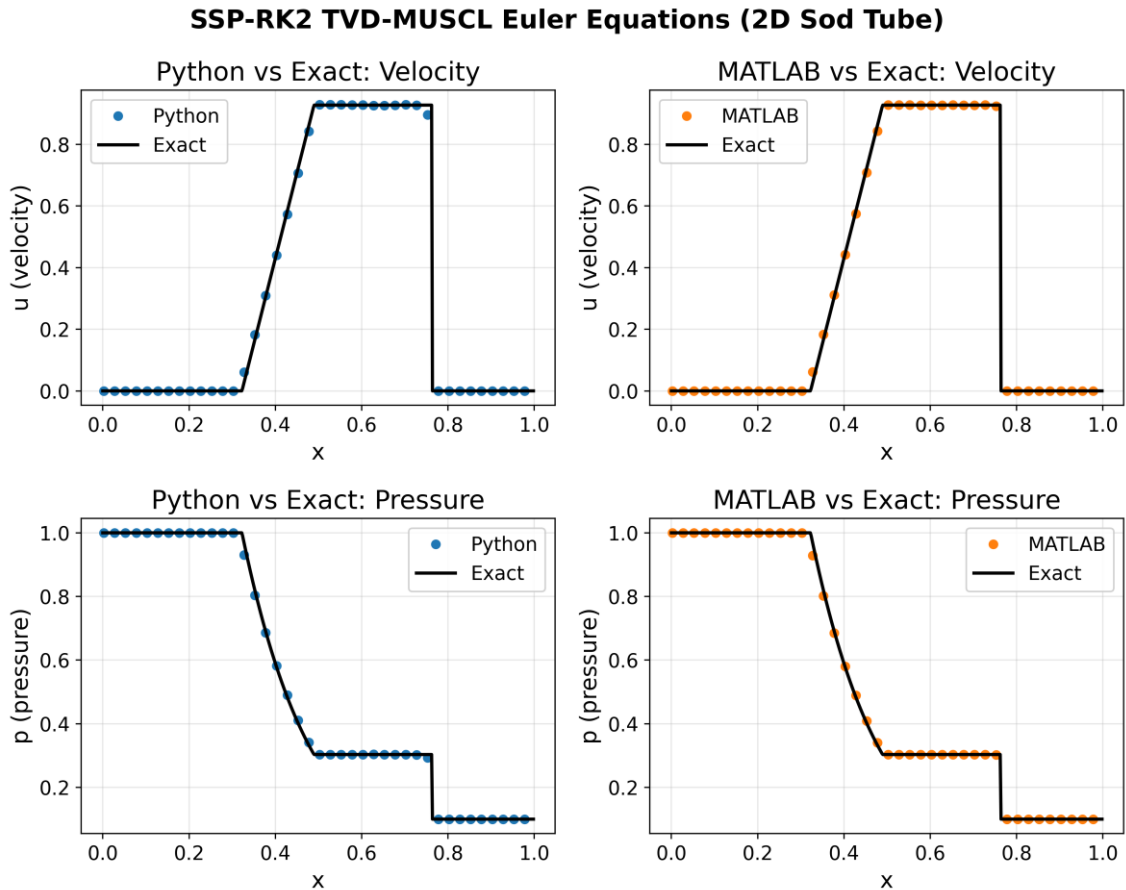


Figure 8.1-2 Sod's shock tube results; Velocity and Pressure visual comparisons.

As a takeaway, the Sod test indicates that the Python implementation reproduces the MATLAB baseline to sub-percent average accuracy for density, pressure, and energy (≈ 0.15 - 0.29%), and to $\sim 2\%$ for velocity (Table 8.1–1), with pointwise maxima concentrated at discontinuities. Relative to the exact Riemann solution, average differences are $\approx 1\%$ for density and pressure and $< 1\%$ for energy, with a larger percentage reported for velocity due to near-zero denominators in quiescent zones (Table 8.1–2). Visually, Figure 8.1-1 and Figure 8.1-2 corroborate the numerical

summaries: close alignment away from discontinuities and localized deviations at sharp features.

8.1.2 Circular blast results

This test is a quick qualitative check for symmetry and absence of artefacts. No formal acceptance criterion is asserted here. Since both integrations are visually indistinguishable from each other the results are only displayed once, at a specific point in time during the simulation for clarity purposes, shown as Figure 8.1-3 [40].

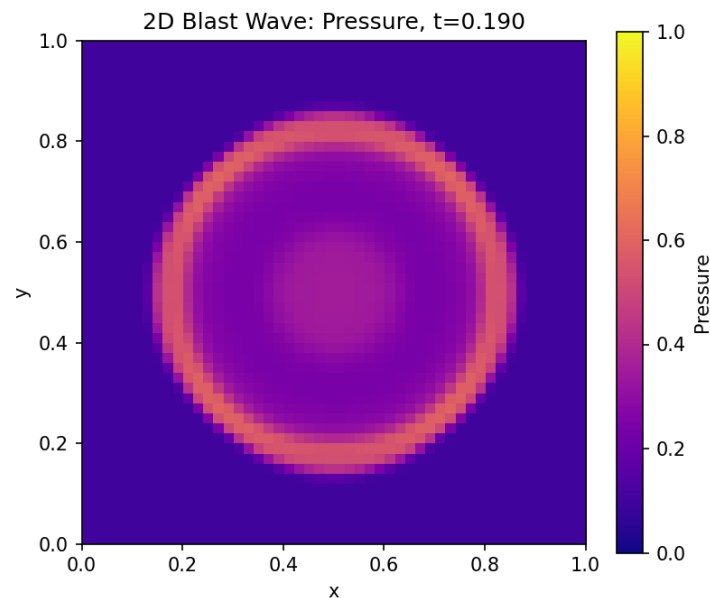


Figure 8.1-3 Circular blast results at time 0.19 seconds.

The baseline and vectorized snapshots are visually consistent: circular symmetry is retained, the expanding shock ring is clean and coherent, and no directional artefacts are observed. When viewed as a sequence, the ring steepens and then rebounds on itself, generating secondary rings that propagate inward and outward, a water-droplet-like pattern consistent with blast-wave dynamics.

8.1.3 Vectorization benchmark results

Figure 8.1-4 shows execution time versus grid size for three code paths (Python non-vectorized, Python vectorized, MATLAB) and the corresponding speed-up ratios relative to the Python vectorized path [40]. Two patterns are clear:

- Algorithmic parity of the baselines. The Python non-vectorized and MATLAB curves are almost coincident across all sizes, indicating comparable baseline implementations under identical initial conditions.
- Substantial, size-robust acceleration from vectorization. The vectorized path lowers wall time by a large, nearly size-independent factor once the grids are moderately large. Speed-up rises from roughly 6–7 $\times$ on very small grids to $\approx 14\times$ at mid sizes and then stabilizes around $\sim 17\text{--}18\times$ for the larger grids. Minor oscillations ($\pm$ a few percent) are typical of cache/OS variability and do not change the overall conclusion.

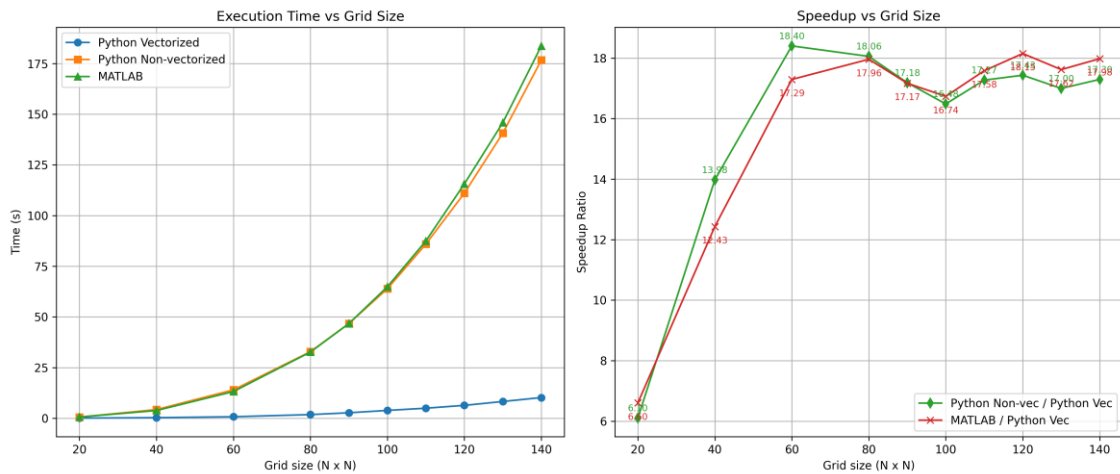


Figure 8.1-4 Vectorization benchmark results; MATLAB; pre-vectorization and post-vectorization execution times alongside vectorization speedup.

Taken together, the benchmarks demonstrate that the optimized vectorized residual preserves the baseline behaviour while delivering an order-of-magnitude reduction in wall time on the problem sizes of interest. The scaling of all three curves is consistent with the expected quadratic growth in work with grid size; vectorization primarily reduces the constant factor by eliminating Python-level looping and improving data locality.

8.2 Chemistry test results

This section reports the **outcomes** of the Chemistry tests introduced in the prior chapter. Each subsection gives a brief orientation to the test, then presents what is reported (plots/tables) and how to read them. Method intent and set-up details are not repeated here; see Chapter 7 for those.

8.2.1 Thermodynamic conversion parity

The results are presented in **Table 8.x–1**, which lists the node temperature T , the internal energies from JAX and Cantera (e_{JAX} , e_{CT}) and their difference, followed by the temperatures recovered from those energies (T_{JAX} , T_{CT}) and their difference. Read the table row-wise: equality in the forward values demonstrates matching NASA-7/EOS evaluations; equality in the recovered temperatures demonstrates a consistent and numerically stable inversion [40].

T (K)	e_{JAX} (J/kg)	e_{CT} (J/kg)	Diff (J/kg)	T_{JAX} (K)	T_{CT} (K)	Diff (K)
300.00	-10491638.60	-10491638.60	0.00	300.00	300.00	0.00
500.00	-10179890.30	-10179890.30	0.00	500.00	500.00	0.00
800.00	-9673749.00	-9673749.00	0.00	800.00	800.00	0.00
1000.00	-9306014.90	-9306014.90	0.00	1000.00	1000.00	0.00
1200.00	-8912904.80	-8912904.80	0.00	1200.00	1200.00	0.00
1500.00	-8278116.10	-8278116.10	0.00	1500.00	1500.00	0.00
2000.00	-7118636.40	-7118636.40	0.00	2000.00	2000.00	0.00
2500.00	-5862332.70	-5862332.70	0.00	2500.00	2500.00	0.00

Table 8.2–1 Thermodynamic conversion parity results ($T \leftrightarrow e$).

For each $T \in \{300, 500, 800, 1000, 1200, 1500, 2000, 2500\}$:

- $e_{\text{JAX}}=e_{\text{CT}}$ to the precisions printed by the test; **maximum energy difference = 0.0 J/kg (0.000000%)**.
- $T_{\text{JAX}}=T_{\text{CT}}$ to the precisions printed by the test; **maximum round-trip temperature error = 0.00 K**.

The zero differences reported for all grid points indicate that the **NASA-7 property evaluation** (cp, h, s) and **ideal-gas EOS closure** are implemented consistently with Cantera and that the **temperature–energy inversion** is well-conditioned over the operating range tested. In practical terms, this removes the thermodynamic layer as a confounding factor: subsequent discrepancies observed in reactive integrations can be attributed to **kinetics** and **time-integration settings** rather than to property evaluation or the $T \leftrightarrow e$ mapping. Within the resolution of the report, the thermodynamic module achieves **machine-precision parity** with Cantera in both directions ($T \rightarrow e$ and $e \rightarrow T$).

8.2.2 Instantaneous kinetics parity

Figure 8.2.2-1 compares the instantaneous species production rates for an H_2 - O_2 mixture (H_2 :10%, O_2 :25%, H_2O :65% by mass) at standard combustion conditions ($T=1500K$, $p=1.2kg/m^3$). The comparison includes both absolute rates and relative differences for the major and minor species in the mechanism [40].

Species	JAX Rate (kmol/m ³ /s)	Cantera Rate (kmol/m ³ /s)	Absolute Difference	Relative Difference (%)
H_2	-4.440e-01	-4.439e-01	7.523e-05	0.0169
O_2	-4.533e-01	-4.533e-01	1.671e-11	0.0000
H	4.442e-01	4.440e-01	1.505e-04	0.0339
O	1.640e-07	1.640e-07	6.046e-18	0.0000
OH	9.448e-03	9.448e-03	3.484e-13	0.0000
HO_2	4.533e-01	4.533e-01	1.671e-11	0.0000
H_2O	-9.448e-03	-9.448e-03	3.484e-13	0.0000

Table 8.2.2-1: Comparison of instantaneous species production rates between JAX and Cantera implementations for a stoichiometric H_2 - O_2 mixture at $T=1500K$, $p=1.2kg/m^3$

The results demonstrate exceptional numerical agreement between the JAX and Cantera implementations. For all species, relative differences remain below

0.04%, with most species showing differences at or near machine precision. The largest absolute difference occurs for atomic hydrogen (H) at 1.505×10^{-4} kmol/m³/s, representing a relative difference of only 0.0339%. The overall normalized RMS difference across all species is 0.0187%, indicating nearly perfect agreement between the implementations.

Particularly significant is the energy source term comparison, which shows excellent agreement (0.030% difference) between JAX (8.865×10^7 J/kg/s) and Cantera (8.862×10^7 J/kg/s). This confirms that not only individual species rates but also the overall energy release rate, critical for accurately capturing the coupling between chemistry and fluid dynamic, is reproduced with high fidelity under the tested specific conditions.

8.2.3 Comprehensive Chemical Kinetics Validation Matrix

This section presents the results of the comprehensive chemical kinetics validation test that evaluates the JAX implementation against the Cantera reference standard. The test spans a multi-dimensional parameter space that covers temperatures from 600K to 2000K and various mixture compositions relevant to hydrogen combustion applications [40].

The validation matrix in Table 8.2–2 presents the quantitative comparison between JAX and Cantera across 18 distinct thermodynamic states, reporting both the maximum species-specific relative differences and the energy source term differences.

Mixture	Temperature (K)	Max Rel Diff (%)	Energy Diff (%)
Lean H ₂ /O ₂	600	0.0000	87.0524
Lean H ₂ /O ₂	800	0.0038	0.0032
Lean H ₂ /O ₂	1000	0.0000	0.0000
Lean H ₂ /O ₂	1200	0.0004	0.0004
Lean H ₂ /O ₂	1500	0.0193	0.0153
Lean H ₂ /O ₂	2000	0.7455	0.3295
Stoichiometric H ₂ /O ₂	600	0.0000	334.3436

Stoichiometric H ₂ /O ₂	800	0.0025	0.0022
Stoichiometric H ₂ /O ₂	1000	0.0000	0.0000
Stoichiometric H ₂ /O ₂	1200	0.0008	0.0007
Stoichiometric H ₂ /O ₂	1500	0.0339	0.0302
Stoichiometric H ₂ /O ₂	2000	1.2684	0.9474
Rich H ₂ /O ₂	600	0.0000	555.8912
Rich H ₂ /O ₂	800	0.0023	0.0020
Rich H ₂ /O ₂	1000	0.0000	0.0000
Rich H ₂ /O ₂	1200	0.0018	0.0017
Rich H ₂ /O ₂	1500	0.0752	0.0679
Rich H ₂ /O ₂	2000	2.7331	2.1972

Table 8.2–2 Comprehensive validation matrix comparing JAX and Cantera implementations across multiple temperatures and compositions.

The results reveal several key patterns:

1. **Temperature-dependent accuracy:** The agreement between JAX and Cantera implementations shows a systematic pattern with temperature. At moderate temperatures (800K-1500K), maximum relative differences remain extremely small (<0.1% for most cases). At the highest temperature tested (2000K), the differences increase somewhat but remain well within acceptable bounds (maximum 2.73% for the rich mixture).
2. **Composition effects:** The rich H₂/O₂ mixture shows slightly larger differences than lean and stoichiometric mixtures, particularly at 2000K. This indicates that the more complex reaction pathways in fuel-rich conditions create somewhat more challenging conditions for numerical consistency.
3. **Species-specific patterns:** The largest differences consistently occur in the atomic hydrogen (H) production rates, with relative differences reaching 2.73% in the most extreme case (rich mixture, 2000K). Other species show considerably smaller differences.

4. **Special case at 600K:** The large percentage differences in energy source terms (reaction's d_e/d_t) at 600K (87.05%, 334.34%, 555.89%) are misleading, as they occur where absolute rates are extremely small ($< 10^{-12}$ kmol/m³/s). The absolute differences remain negligible in these cases, but small denominators magnify the percentage metrics.

This comprehensive validation demonstrates that the JAX implementation reproduces the Cantera reference calculations with high fidelity across the entire parameter space relevant to hydrogen combustion. The mean relative differences across all species and conditions remain below 0.1% for most conditions, with maximum differences reaching 2.73% only in the most extreme case (rich mixture at 2000K).

These results should confirm that any differences observed in integrated chemical kinetics simulations between JAX and Cantera implementations arise not from discrepancies in the instantaneous rate calculations but from differences in the integration methods. This validation establishes a solid foundation for the use of the JAX implementation in reactive flow simulations, with chemical kinetics accuracy comparable to established reference tools.

8.2.4 JAX vs Cantera scaling benchmark

This section presents the computational performance results comparing our JAX-based chemistry implementation against the Cantera reference implementation. The benchmark quantifies the scaling characteristics and performance advantages of the vectorized approach as problem size increases, using the Kvaerno3/Euler stiff solver for the JAX implementation [40].

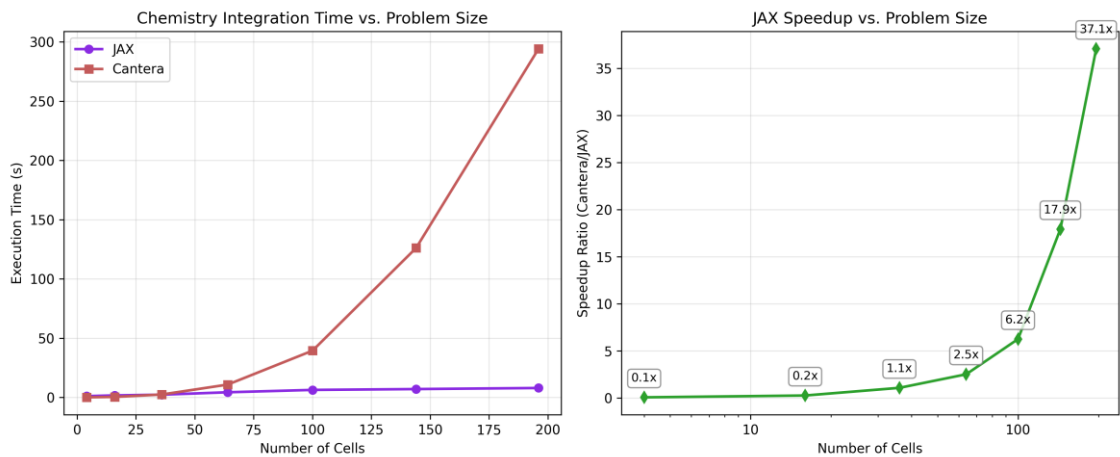


Table 8.2–3 Figure 8.2.4.1: Performance comparison between JAX (purple) and Cantera (desaturated red) implementations showing execution time vs. problem size (left) and speedup vs. problem size (right).

Building on the methodology described in the previous chapter, the benchmark evaluates both implementations across grid sizes ranging from 2×2 to 14×14 cells, measuring execution time, speedup ratios, and solution accuracy for hydrogen-oxygen combustion at 1500K initial temperature.

Grid Size	Cells	JAX Time (s)	Cantera Time (s)	Speedup
2×2	4	1.0748	0.0586	0.05×
4×4	16	1.6822	0.4168	0.25×
6×6	36	2.3004	2.4422	1.06×
8×8	64	4.3277	10.8293	2.50×
10×10	100	6.3194	39.4350	6.24×
12×12	144	7.0364	126.0813	17.92×
14×14	196	7.9363	294.1651	37.07×

Table 8.2–4 Performance Comparison between JAX and Cantera Implementations.

Several important observations emerge from the benchmark results:

Several important observations emerge from the benchmark results:

1. **Computational Efficiency:** For larger simulations, the JAX implementation offers substantial computational savings, potentially enabling higher-resolution simulations or faster time-to-solution for the same computational resources.
2. **Scalability:** The superior scaling characteristics of the JAX implementation suggest that its advantages will become even more pronounced for larger problem sizes typical of real-world applications.
3. **Small-Scale Considerations:** For very small problems, the standard Cantera implementation remains more efficient, suggesting that hybrid approaches could be beneficial in certain scenarios.
4. **Resource Utilization:** JAX's ability to leverage GPU acceleration (not explicitly tested in this benchmark) offers additional performance potential beyond the CPU-based comparisons shown here.

The benchmark results have significant implications for reactive flow simulations:

1. **Crossover Point:** For small problem sizes (grid sizes 2×2 and 4×4), Cantera outperforms the JAX implementation. This is expected due to the overhead associated with JAX's just-in-time compilation setup, which incurs fixed costs regardless of problem size.
2. **Performance Inflection:** At the 6×6 grid (36 cells), the JAX implementation begins to outperform Cantera, achieving a modest speedup of $1.06\times$. This represents the crossover point where the optimization benefits begin to outweigh the initialization overhead.
3. **Scaling Efficiency:** JAX demonstrates superior scaling efficiency (0.77) compared to Cantera's super-linear scaling (2.19). This indicates that Cantera's computational cost grows more rapidly with problem size, while JAX maintains relatively efficient scaling.

4. **Large-Scale Performance:** At the largest tested grid size (14×14, 196 cells), JAX achieves a remarkable 37.07× speedup over Cantera. This dramatic difference illustrates JAX's effectiveness for larger-scale chemistry calculations.

The benchmark results have significant implications for reactive flow simulations:

1. **Computational Efficiency:** For larger simulations, the JAX implementation offers substantial computational savings, potentially enabling higher-resolution simulations or faster time-to-solution for the same computational resources.
2. **Scalability:** The superior scaling characteristics of the JAX implementation suggest that its advantages will become even more pronounced for larger problem sizes typical of real-world applications.
3. **Resource Utilization:** JAX's ability to leverage GPU acceleration (not explicitly tested in this benchmark) offers additional performance potential beyond the CPU-based comparisons shown here.

The JAX-based chemistry implementation demonstrates compelling performance benefits for moderate to large problem sizes, with speedup factors increasing dramatically as the problem size grows. While there is a performance penalty for very small problems, the crossover point occurs at relatively modest grid sizes (6×6, 36 cells).

The benchmark confirms that the JAX implementation achieves its primary objective: enabling significantly faster chemical integration for multi-dimensional reactive flow simulations without compromising accuracy. The scaling characteristics suggest that these benefits will extend to production-scale simulations with thousands or millions of cells, making this approach particularly valuable for computationally intensive applications like detonation simulations.

8.2.5 Divergence vs initial condition and runtime

Building on the methodology described in the previous section, this section presents a detailed analysis of the numerical stability results between JAX and Cantera implementations. The analysis focuses on identifying specific regions and conditions where numerical behaviour diverges and providing quantitative assessment of these differences [40].

Temperature Divergence vs Initial Temperature vs Simulation Time (P=1atm)

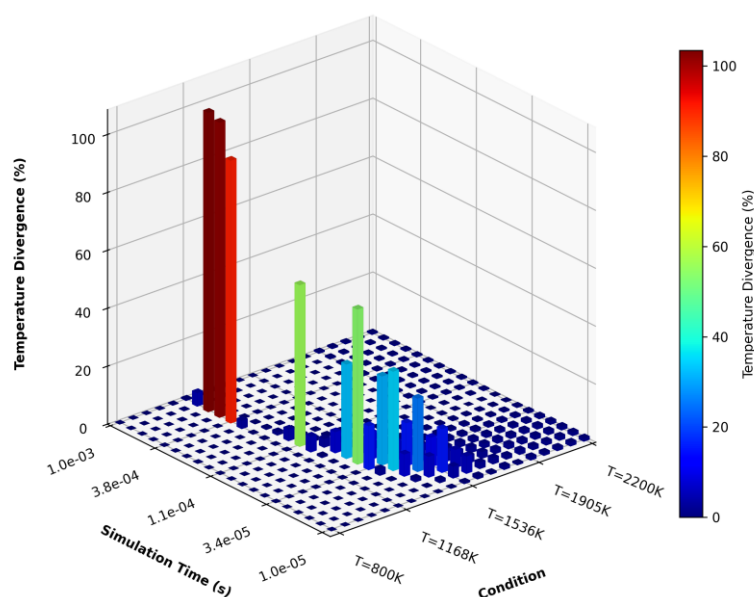


Figure 8.2-1 Temperature divergence between JAX and Cantera implementations across varying initial temperatures and integration times at constant pressure (1 atm).

The temperature variation analysis shown in Figure 8.2-1, formalized in Table 8.2–5 reveals several important patterns:

1. **Stable regions:** As visible in the lower left region of the figure, for initial temperatures below ~1200K, the JAX implementation maintains excellent agreement with Cantera (divergence < 0.1%) across all integration timeframes tested.

2. **Transition zone:** In the range of 1300-1400K, shown in the middle section, the JAX implementation begins to show sensitivity to integration time, with moderate divergence (0.1-5%) appearing at longer integration times.
3. **Critical divergence regions:** The most pronounced divergence, appearing as tall red bars, occurs in specific combinations of initial temperature and integration time:
 - At $T_0 = 1315\text{K}$ with simulation time $4.28 \times 10^{-5}\text{ s}$, showing a maximum divergence of 53.61%.
 - At $T_0 = 1242\text{K}$ with simulation time $1.13 \times 10^{-4}\text{ s}$, showing a maximum divergence of 55.87%.
 - At $T_0 = 1168\text{K}$ with simulation times around $3.79 \times 10^{-4}\text{ s}$, showing a maximum divergence of 90.78% and reaching as high as 102% at longer integration times.
4. **Recovery at long timescales:** For most initial temperatures in the range 1350-1900K, the divergence peaks at intermediate timescales and then diminishes at longer integration times, suggesting the JAX and Cantera implementations ultimately converge to similar equilibrium states.

Initial T (K)	Peak Divergence (%)	Simulation Time (s)	Comments
800-1094	< 0.1	All times	Excellent agreement throughout
1168	102.0	4.83×10^{-4}	Extreme divergence at longer times
1242	55.9	1.13×10^{-4}	High divergence at specific time
1315	53.6	4.28×10^{-5}	High divergence at specific time
1389	34.4	2.64×10^{-5}	Moderate divergence at short time
1463	25.4	2.07×10^{-5}	Moderate divergence at short time
1536	15.2	1.62×10^{-5}	Lower peak divergence
1684-2200	< 2.5	All times	Good agreement at higher temperatures

Table 8.2–5 Peak Temperature Divergence (%) at Selected Initial Temperatures

Temperature Divergence vs Pressure vs Simulation Time (T=1500K)

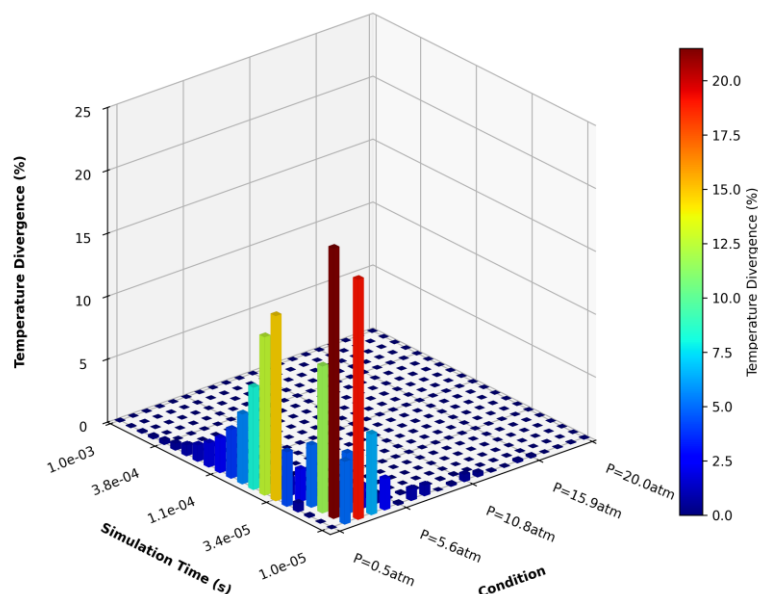


Figure 8.2-2 Temperature divergence between JAX and Cantera implementations across varying initial pressures and integration times at constant temperature (1500K).

The pressure variation analysis illustrated in Figure 8.2-1, formalized in Table 8.2–6, shows complementary but distinct patterns:

1. **Stable regions:** As evident from the flat regions in the figure, for most pressure conditions above 4 atm, the JAX implementation maintains excellent agreement with Cantera (divergence < 0.1%) across all integration timeframes.
2. **Low-pressure sensitivity:** The largest divergences, visible as the tallest bars, occur at low pressures (0.5-2.6 atm), with maximum divergence of about 21.5% occurring at 1.5 atm and integration time of 1.27×10^{-5} s.
3. **Time-dependent behaviour:** Unlike the temperature variation cases in Figure 8.2-1, pressure-related divergences show a more consistent pattern of decreasing with increasing integration time, suggesting better convergence at equilibrium.

4. **Critical pressure threshold:** The most significant divergences appear around 1.5-3 atm, suggesting a threshold where the solver sensitivity is highest, possibly related to ignition delay characteristics at these conditions.

Initial P (atm)	Peak Divergence (%)	Simulation Time (s)	Comments
0.5	14.7	3.36×10^{-5}	Moderate divergence at short times
1.5	21.5	1.27×10^{-5}	Highest divergence overall
2.6	19.1	1.00×10^{-5}	Significant divergence at shortest time
3.6	6.4	1.00×10^{-5}	Lower divergence
4.6-20.0	< 2.5	All times	Good agreement at higher pressures

Table 8.2–6 Peak Temperature Divergence (%) at Selected Initial Pressures.

The observed divergence patterns could be attributed to several numerical and physical factors:

1. **Stiffness in transition regions:** The highest divergences visible most likely correspond to thermochemical conditions where the ODE system becomes extremely stiff due to rapidly accelerating reaction rates during ignition events. For example, around 1200-1350K, the hydrogen-oxygen system transitions from slow to fast chemistry, creating numerical challenges.
2. **Solver strategy differences:** The JAX implementation uses the Kvaerno3/Euler stiff solver, while Cantera employs CVODE with adaptive time stepping. These different numerical approaches handle extremely stiff regions differently, explaining the distinct divergence patterns seen:
 - JAX: Fixed step size may miss or overreact to rapid chemical transitions
 - Cantera: Adaptive timesteps can better resolve rapid transitions but may follow a different numerical trajectory
3. **Ignition timing sensitivity:** Near ignition thresholds, small differences in numerical handling can lead to significant differences in ignition timing. If

ignition occurs at slightly different points in the integration for JAX versus Cantera (e.g., one time step earlier or later), temperature predictions will diverge dramatically at fixed comparison times.

4. **Integration method impact:** The matrix data shows that for certain conditions (e.g., $T=1168\text{K}$), divergence increases with longer integration times, suggesting fundamental differences in how the solvers handle specific chemical pathways or equilibrium states.
5. **Pressure-temperature interaction:** The comparative analysis of both figures shows that low pressures (0.5-3 atm) combined with moderate temperatures (around 1500K) create particularly challenging conditions for numerical consistency.

This detailed analysis of divergence patterns between JAX and Cantera chemistry implementations, reveals specific regimes where numerical behaviour differs significantly. The most pronounced divergences occur in narrow bands of temperature-time or pressure-time conditions associated with rapid chemical transitions.

The findings highlight that while the JAX chemistry implementation performs exceptionally well across many conditions and offers significant computational speed advantages, there remain important areas requiring further investigation.

9 Conclusions

This work set out to establish a lean, reproducible baseline for detonation-oriented computation by implementing a shock-capturing, inviscid Euler solver in vectorized Python and developing and cross-checking a standalone thermochemistry module against a community reference. All conclusions below reflect what can be asserted with confidence given the project's scope and the verification stance adopted during the thesis' development.

9.1 CFD solver fidelity from canonical problems

Across the canonical Sod shock tube and a 2-D circular blast, the code exhibits the hallmarks of a correctly implemented Godunov-type finite-volume method: sharp shocks without non-physical oscillations, clean contacts, and smooth rarefactions. In Sod, the Python solver tracks both the exact Riemann solution and a MATLAB baseline closely on all conserved variables, with the largest pointwise deviations confined to narrow layers at discontinuities, precisely where limiter-based second-order schemes are expected to trade accuracy for monotonicity. The qualitative blast snapshot reinforces this picture: rotational symmetry is maintained, and no grid-alignment artefacts appear, which is a sensitive visual check of flux formulation and reconstruction symmetry. These findings, taken together, indicate that the spatial discretization and time integration behave as intended on standard code-verification problems.

9.2 Methodological soundness of MUSCL–HLLC–SSP-RK2

The observed behaviour aligns with the theoretical properties of the chosen numerics. MUSCL reconstruction with TVD limiting provides non-oscillatory second-order resolution away from discontinuities; the HLLC family's robust wave-speed estimates contribute to positivity and stability in the presence of strong rarefactions and shocks; and SSP Runge–Kutta time stepping preserves the TVD property of the forward-Euler update under a CFL constraint. The aggregate effect is exactly what the results show: visually coincident solution overlays in smooth regions and bounded, localized differences across shocks/contacts.

9.3 Performance: CFD solver (vectorization)

Vectorizing the residual and reconstruction operators produces large, size-robust gains while preserving the baseline algorithm. The three-curve benchmark (Python non-vectorized, Python vectorized, MATLAB) shows two stable patterns that matter for feasibility:

- **Baseline parity.** The Python non-vectorized and MATLAB timings are nearly coincident across sizes, indicating like-for-like algorithms and isolating vectorization as the differentiator rather than hidden model changes.
- **Sustained speedup at usable sizes.** Speedup levels off around ~17–18× on the larger grids tested. All three paths exhibit the expected quasi-quadratic growth in work with grid size; vectorization primarily lowers the constant factor by removing Python-level looping and improving memory locality.

These gains turn larger domains and longer integrations from impractical to routine under the same resource envelope, precisely what is required to resolve azimuthal wave structure and multi-cycle behaviour in detonation-type configurations. In short, the vectorized CFD path is the one that scales to the mesh sizes that matter.

9.4 Performance: chemistry integration (JAX vs. Cantera, JIT-compiled)

The batched JAX chemistry shows a complementary but distinct performance profile driven by just-in-time (JIT) compilation and fused array kernels:

- **Crossover behaviour.** For tiny problems the JIT overhead dominates (2×2: 1.0748 s JAX vs 0.0586 s Cantera). The curves cross by 6×6 (36 cells) where JAX is modestly faster (2.3004 s vs 2.4422, speedup; 1.06×).

- **Accelerating advantage with size.** Beyond the crossover point, speedups grow steeply: 8×8: 2.50×; 10×10: 6.24×; 12×12: 17.92×; 14×14: 37.07× (7.9363 s JAX vs 294.1651 s Cantera). The experiment had to stop before any JAX plateau was observed because the reference runs became prohibitively slow, i.e., the speedup curve had not levelled out within the tested range.
- **What the shape means.** This is the expected signature of JIT-compiled array programs: a fixed compilation/setup cost that penalizes tiny cases, followed by amortization and sustained efficiency as batched reactors grow. In contrast, general-purpose per-reactor ODE solves incur overheads that scale less favourably when the batch is large.

For the large reactor counts needed by operator-split reactive CFD at RDE-relevant resolutions, the JAX chemistry becomes the only practical option among those tested. Since instantaneous rates and thermodynamic round-trips already match the reference to tight tolerances, these performance gains come without compromising chemical fidelity.

9.5 Thermochemical layer correctness and independence

Round-trip thermodynamics ($T \leftrightarrow e$) match Cantera to print precision across a wide temperature range, and instantaneous species and energy source rates agree to within small relative differences over representative lean/stoichiometric/rich hydrogen–oxygen states. Because the module draws on standard NASA 7-coefficient thermochemical polynomials, the same primary data used by widely adopted tools, this parity indicates that property evaluation and kinetics bookkeeping are correctly implemented. In practical terms, the chemistry layer no longer confounds diagnosis: when discrepancies appear in time-integrated reactors, they can credibly be attributed to integration strategy and stiffness handling rather than to thermodynamic closure or algebraic rate evaluation.

9.6 Where and why numerical trajectories diverge

The divergence maps over initial temperature, pressure, and integration time identify narrow bands of larger JAX-Cantera differences that coincide with ignition-threshold regimes. Outside those bands, agreement remains tight. Within them, rates accelerate rapidly, and the ODEs become numerically stiff; in such transients, small differences in the integrators' internal time histories can produce large discrepancies at fixed comparison times.

On the JAX side the runs employed Difffrax's Kvaerno3: an explicit embedded Runge–Kutta scheme (Dormand–Prince) that provides a built-in error estimate and supports adaptive step-size control via a PID-type controller [55]. On the Cantera side the reactor network is advanced by SUNDIALS CVODE: a variable-order, variable-step multistep integrator (Adams for non-stiff; BDF for stiff) with adaptive error control. Both frameworks therefore adapt their step sizes, but they do so with different families of methods (one-step RK vs. multistep LMM), different error controllers, and different internal acceptance/rejection logic.

The patterns observed are consistent with these algorithmic differences interacting with stiffness near ignition (e.g., each method may register the onset of rapid heat release one internal step earlier or later, leading to large temperature gaps at a chosen output time). However, the experiments here do not isolate causation. Particularly, controlled ablations were not performed to evaluate the following: sweep tolerances and initial step sizes on both sides; swap the JAX solver to a stiff, implicit DIRK family under otherwise identical conditions; or align ignition detection via event-based stopping and compare ignition-delay times rather than raw state at fixed times. Absent those controls, it is more rigorous to present these maps as a robustness envelope: very strong agreement across broad swaths of state space, with well-localized pockets where solver strategy and stiffness management dominate the numerical trajectory.

9.7 Scope and limits of what is concluded

The conclusions above speak to verification; namely, that the implemented algorithms solve the intended equations correctly on problems where trusted references exist. They do not claim validation against experiments or fully coupled detonation physics. In-line coupling of detailed chemistry to the flow solver were designed but not exercised in the production CFD runs; the chemistry results arise from decoupled multi-reactor studies. Within these acknowledged limits, the body of evidence supports three overarching statements:

- The MUSCL–HLLC–SSP-RK2 Euler solver is implemented correctly and behaves robustly on canonical tests.
- Vectorization delivers substantial, size-robust speedups without compromising fidelity.
- The thermochemistry module reproduces the reference tool across most of the relevant state space, with divergence confined to stiff ignition bands where solver strategy, not algebraic rates, dominates outcomes. This is consistent with a verification-first posture for building credible CFD software.

10 Future work

This thesis established an inviscid, shock-capturing baseline and an independently verified thermochemistry module. The next steps fall into Physics fidelity and coupling, and numerical robustness and performance at scale. The items below are prioritized to deliver the largest credibility and capability gains for detonation-oriented simulations.

10.1 Reactive coupling at the solver core

The immediate step is to move from an inviscid baseline to a fully reactive solver in which detailed chemistry advances in lock-step with the flow. The coupling should follow a conservative operator-splitting design (Strang-type) so that mass fractions and total energy remain consistent across convective and reactive sub-steps. Because ignition onsets are sharp, outputs should be synchronized by event-based ignition detection rather than by fixed sampling times. Work begins on uniform grids and the same boundary conditions used in verification, then extends to inflow/outflow set-ups once conservation and stability are demonstrated.

10.2 Numerical robustness in ignition bands

Ignition-threshold regimes make the ODEs extremely stiff and are the main source of solver sensitivity. The near-term goal is to establish a robust envelope of agreement across integrator families and tolerances. This involves re-running the existing divergence maps while sweeping absolute/relative tolerances and initial step sizes, and comparing integral diagnostics (e.g., ignition delay, equilibrium composition) rather than instantaneous snapshots. On the JAX side, tests should include stiff-capable integrators. The result should be a documented set of settings that behave predictably across the troublesome temperature-pressure bands.

10.3 Performance and scalability

Two complementary tracks extend the performance gains already observed.

- CFD path. Apply just-in-time compilation to the Euler residual and reconstruction while preserving memory locality; profile cache behaviour, align data layouts with face-flux usage, and thread halo exchanges and reductions.
- Chemistry path. Push the batch-parallel JAX integrators to larger reactor counts and GPU backends. Pre-compile hot graphs, minimize host–device transfers, and benchmark implicit stiff solvers for ignition-band cases. Track wall-time vs. problem size, speedup with/without JIT warm-up, and produce simple roofline-style plots to expose compute vs. memory limits.

These steps convert the observed mid-scale speedups into production-scale throughput, which is a prerequisite for credible RDE studies.

10.4 Reactive benchmarks and targeted validation

Before attempting full RDE configurations, the coupled solver will be anchored against tabulated CJ speeds and ZND profiles for standard mixtures, and against curated ignition-delay datasets where available. Claims here stay deliberately narrow: the aim is to establish traceable reference points and quantify agreement bands, not to exhaustively validate detonation physics.

References

- [1] V. Raman, S. Prakash, and M. Gamba, "Nonidealities in Rotating Detonation Engines," *Annu Rev Fluid Mech*, vol. 55, no. Volume 55, 2023, pp. 639–674, Jan. 2023, doi: 10.1146/ANNUREV-FLUID-120720-032612/CITE/REFWORKS.
- [2] H. D. Perkins and D. E. Paxson, "Summary of Pressure Gain Combustion Research at NASA," 2018, Accessed: Aug. 24, 2025. [Online]. Available: <http://www.sti.nasa.gov>
- [3] "ROTATING DETONATION ENGINES (RDE) – Air Force Research Laboratory." Accessed: Aug. 19, 2025. [Online]. Available: <https://afresearchlab.com/technology/rotating-detonation-engines-rde/>
- [4] D. Perkins and D. Paxson, "Rotating Detonation Research at NASA Motivation for Airbreathing Rotating Detonation Research Potential Impact of PGC on CFM56-7B27," 2023.
- [5] Y. Huang, H. Xia, X. Chen, Z. Luan, and Y. You, "Shock dynamics and expansion characteristics of an aerospike nozzle and its interaction with the rotating detonation combustor," *Aerosp Sci Technol*, vol. 117, p. 106969, Oct. 2021, doi: 10.1016/J.AST.2021.106969.
- [6] M. L. Fotia, F. Schauer, T. Kaemming, and J. Hoke, "Experimental Study of the Performance of a Rotating Detonation Engine with Nozzle," <https://doi.org/10.2514/1.B35913>, vol. 32, no. 3, pp. 674–681, Dec. 2015, doi: 10.2514/1.B35913.
- [7] S. Blanes, F. Casas, A. Murua, K. Zientziak, and A. A. Salla, "Splitting methods for differential equations," pp. 1–161, 2024, doi: 10.1017/S0962492923000077.
- [8] G. Strang, "On the Construction and Comparison of Difference Schemes," <https://doi.org/10.1137/0705041>, vol. 5, no. 3, pp. 506–517, Jul. 2006, doi: 10.1137/0705041.

- [9] G. A. Sod, "A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws," *J Comput Phys*, vol. 27, no. 1, pp. 1–31, Apr. 1978, doi: 10.1016/0021-9991(78)90023-2.
- [10] C. R. Harris *et al.*, "Array programming with NumPy," *Nature* 2020 585:7825, vol. 585, no. 7825, pp. 357–362, Sep. 2020, doi: 10.1038/s41586-020-2649-2.
- [11] "DiffraX." Accessed: Sep. 08, 2025. [Online]. Available: <https://docs.kidger.site/diffrax/>
- [12] "JAX: High performance array computing — JAX documentation." Accessed: Sep. 08, 2025. [Online]. Available: <https://docs.jax.dev/en/latest/index.html>
- [13] D. G. Goodwin *et al.*, "Cantera: An Object-oriented Software Toolkit for Chemical Kinetics, Thermodynamics, and Transport Processes," *zndo*, Feb. 2021, doi: 10.5281/ZENODO.4527812.
- [14] M. Smale and T. Jayne, "Maize in Eastern and Southern Africa : ‘ Seeds ’ of Success in Retrospect," *Food Policy*, no. 97, pp. 1–79, 2003, Accessed: Sep. 11, 2025. [Online]. Available: <https://archive.org/details/detonationtheory0000fick>
- [15] B. Einfeldt, "On Godunov-Type Methods for Gas Dynamics," <https://doi.org/10.1137/0725021>, vol. 25, no. 2, pp. 294–318, Jul. 2006, doi: 10.1137/0725021.
- [16] B. van Leer, van Leer, and B., "Towards the Ultimate Conservative Difference Scheme. V. A Second-Order Sequel to Godunov’s Method," *JCoPh*, vol. 32, no. 1, pp. 101–136, 1979, doi: 10.1016/0021-9991(79)90145-1.
- [17] P. K. Sweby, "High Resolution Schemes Using Flux Limiters for Hyperbolic Conservation Laws," *SIAM J Numer Anal*, vol. 21, no. 5, pp. 995–1011, 1984.

- [18] S. Gottlieb, C.-W. Shu, and E. Tadmor, “Strong Stability-Preserving High-Order Time Discretization Methods *,” *Society for Industrial and Applied Mathematics*, vol. 43, no. 1, pp. 89–112, 2001, Accessed: Aug. 30, 2025. [Online]. Available: <http://www.siam.org/journals/sirev/43-1/36757.html>
- [19] A. Burcat and B. Ruscic, “Third Millennium Ideal Gas and Condensed Phase Thermochemical Database for Combustion with Updates from Active Thermochemical Tables,” 2005, Accessed: Sep. 11, 2025. [Online]. Available: www.anl.gov.
- [20] B. J. McBride, M. J. Zehe, and S. Gordon, “NASA Glenn Coefficients for Calculating Thermodynamic Properties of Individual Species,” 2002. Accessed: Sep. 11, 2025. [Online]. Available: <http://www.sti.nasa.gov>
- [21] K. E. Niemeyer, N. J. Curtis, and C.-J. Sung, “pyJac: analytical Jacobian generator for chemical kinetics,” *Comput Phys Commun*, vol. 215, pp. 188–203, Feb. 2017, doi: 10.1016/j.cpc.2017.02.004.
- [22] Sponsor, “AIAA G-0 7 7-1 9 9 8 Guide Guide for the Verification and Validation of Computational Fluid Dynamics Simulations,” 1998.
- [23] G. A. Sod, “A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws,” *J Comput Phys*, vol. 27, no. 1, pp. 1–31, Apr. 1978, doi: 10.1016/0021-9991(78)90023-2.
- [24] E. F. Toro, “Riemann solvers and numerical methods for fluid dynamics: A practical introduction,” *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction*, pp. 1–724, 2009, doi: 10.1007/B79761/COVER.
- [25] “Finite Volume Methods for Hyperbolic Problems”.
- [26] “standard pressure,” *The IUPAC Compendium of Chemical Terminology*, Sep. 2008, doi: 10.1351/GOLDBOOK.S05921.
- [27] M. B. Ewing, T. H. Lilley, G. M. Olofsson, M. T. Rätzsch, and G. Somsen, “Standard quantities in chemical thermodynamics Fugacities, activities,

- and equilibrium constants for pure and mixed phases (IUPAC recommendations 1994),” *Pure and Applied Chemistry*, vol. 66, no. 3, pp. 533–552, 1994, doi: 10.1351/PAC199466030533.
- [28] R. Design, “Chemkin Theory Manual Chemkin ® Software,” 2016.
- [29] “Reactions — Cantera 3.2.0a2 documentation.” Accessed: Sep. 06, 2025. [Online]. Available: <https://www.cantera.org/dev/yaml/reactions.html>
- [30] R. G. Gilbert, K. Luther, and J. Troe, “THEORY OF THERMAL UNIMOLECULAR REACTIONS IN THE FALL-OFF RANGE - 2. WEAK COLLISION RATE CONSTANTS.,” *Berichte der Bunsengesellschaft/Physical Chemistry Chemical Physics*, vol. 87, no. 2, pp. 169–177, 1983, doi: 10.1002/BBPC.19830870218.
- [31] B. J. McBride, S. Gordon, A. Cleveland, and M. A. Reno, “NASA Technical Memorandum 4513 Coefficients for Calculating Thermodynamic and Transport Properties of Individual Species”, Accessed: Sep. 07, 2025. [Online]. Available: <https://ntrs.nasa.gov/search.jsp?R=19940013151>
- [32] “Species Thermodynamic Models — Cantera 3.1.0 documentation.” Accessed: Sep. 07, 2025. [Online]. Available: <https://www.cantera.org/3.1/reference/thermo/species-thermo.html>
- [33] “About Python™ | Python.org.” Accessed: Aug. 30, 2025. [Online]. Available: <https://www.python.org/about/>
- [34] “Pricing and Licensing - MATLAB & Simulink.” Accessed: Aug. 30, 2025. [Online]. Available: <https://uk.mathworks.com/pricing-licensing.html>
- [35] “Frequently Asked Questions — scikit-learn 1.7.1 documentation.” Accessed: Aug. 30, 2025. [Online]. Available: <https://scikit-learn.org/stable/faq.html>

- [36] “venv — Creation of virtual environments — Python 3.13.7 documentation.” Accessed: Aug. 30, 2025. [Online]. Available: <https://docs.python.org/3/library/venv.html>
- [37] “User Guide - pip documentation v25.2.” Accessed: Aug. 30, 2025. [Online]. Available: https://pip.pypa.io/en/stable/user_guide/
- [38] “Managing environments — conda 25.7.1.dev29 documentation.” Accessed: Aug. 30, 2025. [Online]. Available: <https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html>
- [39] “Installing Miniconda - Anaconda.” Accessed: Aug. 30, 2025. [Online]. Available: <https://www.anaconda.com/docs/getting-started/miniconda/install>
- [40] “FagioDiFapo/ApproximateRiemannSolversPy: Parial port to python of Riemann solvers using MUSCL and WENO schemes written as short Matlab scripts.” Accessed: Sep. 11, 2025. [Online]. Available: <https://github.com/FagioDiFapo/ApproximateRiemannSolversPy>
- [41] “Run Code Faster With the New MATLAB Execution Engine » Loren on the Art of MATLAB - MATLAB & Simulink.” Accessed: Aug. 31, 2025. [Online]. Available: <https://blogs.mathworks.com/loren/2016/02/12/run-code-faster-with-the-new-matlab-execution-engine/>
- [42] “Structured arrays — NumPy v2.3 Manual.” Accessed: Sep. 01, 2025. [Online]. Available: <https://numpy.org/doc/stable/user/basics.rec.html>
- [43] “Indexing — NumPy v1.13 Manual.” Accessed: Sep. 02, 2025. [Online]. Available: <https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.indexing.html#advanced-indexing>
- [44] “Boundary conditions — Clawpack 5.4.1 documentation.” Accessed: Sep. 08, 2025. [Online]. Available: <https://www.clawpack.org/master/bc.html>

- [45] R. J. LeVeque, "Boundary Conditions and Ghost Cells," *Finite Volume Methods for Hyperbolic Problems*, pp. 129–138, Aug. 2002, doi: 10.1017/CBO9780511791253.008.
- [46] "Boundary conditions — Clawpack 5.4.1 documentation." Accessed: Sep. 08, 2025. [Online]. Available: <https://www.clawpack.org/master/bc.html>
- [47] H. Jasak, A. Jemcov, ~ Zeljko, and T. Tukovi'c, "OpenFOAM: A C++ Library for Complex Physics Simulations," *International Workshop on Coupled Methods in Numerical Dynamics IUC*.
- [48] H. G. Weller *et al.*, "A tensorial approach to computational continuum mechanics using object-oriented techniques," *Computer in Physics*, vol. 12, no. 6, pp. 620–631, Nov. 1998, doi: 10.1063/1.168744.
- [49] B. W. R. Forde, R. O. Foschi, and S. F. Stiemer, "Object-oriented finite element analysis," *Comput Struct*, vol. 34, no. 3, pp. 355–374, 1990, doi: 10.1016/0045-7949(90)90261-Y.
- [50] "How Time Integration of Reactor Networks Works — Cantera 3.2.0a2 documentation." Accessed: Sep. 08, 2025. [Online]. Available: <https://cantera.org/dev/develop/reactor-integration.html>
- [51] "Parallelizing transport property calculations — Cantera 3.1.0 documentation." Accessed: Sep. 08, 2025. [Online]. Available: https://cantera.org/3.1/examples/python/transport/multiprocessing_viscosity.html
- [52] "How to enable multi-threaded solving." Accessed: Sep. 08, 2025. [Online]. Available: <https://groups.google.com/g/cantera-users/c/j3xaH9scZfQ>
- [53] K. Kim, O. H. Díaz-Ibarra, H. N. Najm, J. Zádor, and C. Safta, "TChem: A performance portable parallel software toolkit for complex kinetic mechanisms," *Comput Phys Commun*, vol. 285, p. 108628, Apr. 2023, doi: 10.1016/J.CPC.2022.108628.

- [54] “How to choose a solver - DiffraX.” Accessed: Sep. 08, 2025. [Online]. Available: <https://docs.kidger.site/diffrax/usage/how-to-choose-a-solver/>
- [55] “ODE solvers - DiffraX.” Accessed: Sep. 08, 2025. [Online]. Available: https://docs.kidger.site/diffrax/api/solvers/ode_solvers/
- [56] “Standard for Verification and Validation in Computational Fluid Dynamics and Heat Transfer,” 2009.

Appendix A Ethical approval letter



18 April 2025

Dear Mr Bonet Garcia ,

Reference: CURES/24861/2025

Project ID: 28179

Title: RDE reduced 2D – Detonation tube validation and RDE simulation

We are pleased to inform you that you have successfully declared that your research project is a **Literature Review – based solely on openly available literature which is in the public domain, and you are undertaking desk-based research not involving any other form of data or information.**

You have also confirmed that your project **does not meet** any of the literature review specified exceptions, listed both within the relevant section of the CURES form and below:

1) Your supervisor has requested that you apply for approval through CURES because:

- The journals or data are in a sensitive area (please discuss this with your supervisor)
- The project will be embargoed **i.e. will not be publicly available via the Cranfield library immediately or longer term**

2) Approval is/will be specifically required by another external body e.g. journal publishers

Therefore, **you do not require ethical approval** and your CURES application will be automatically closed.

Please keep a copy of this letter safe, if this exception is in relation to your thesis project, you will need to include a copy with your final thesis submission.

If you have any queries, please contact CURES Support.

We wish you every success with your project.

Regards,

CURES Team

Appendix B RHS Thermochemistry Landmarks

This appendix is meant to show a glance of the underlying thermochemical process described in sections 3, 4 and 6 as implemented in the code [40].

B.1 Mechanism ingestion & dense arrays

B-1 flatten a Cantera mechanism (stoichiometry, rate parameters, third-body efficiencies, falloff/Troe data) into dense arrays usable by batched RHS.

```
MECHANISM = BuildMechanism(YAML_file):
    gas ← Cantera.Solution(YAML_file)
    S ← gas.n_species ; R ← gas.n_reactions

    # NASA-7 thermo per species
    for k in 1..S:
        (Tmid[k], a_hi[k,1:7], a_lo[k,1:7]) ← extract_NASA-7(gas.species[k])

    # Dense stoichiometric matrices
    v_f[R,S] ← 0 ; v_b[R,S] ← 0 ; reversible[R] ← 0
    rxn_type[R] ← 0      # 0: elementary, 1: three-body, 2: falloff
    eff[R,S] ← 1.0      # third-body efficiencies (default 1)
    A[R], b[R], Ea[R] ← 0      # high-P Arrhenius
    A0[R], b0[R], Ea0[R] ← 0   # low-P Arrhenius (falloff)
    troe_a[R], T3[R], T1[R], T2[R] ← 0
    has_falloff[R] ← 0

    for r in 1..R:
        reversible[r] ← (reaction r reversible?)
        fill v_f[r,*] from reactants; fill v_b[r,*] from products
        if reaction r has third_body: set rxn_type[r] = max(rxn_type[r], 1) and eff[r,*] from
efficiencies
        if reaction r has (low_rate, high_rate):
            rxn_type[r] = 2 ; has_falloff[r] = 1
            (A,b,Ea) ← high_rate ; (A0,b0,Ea0) ← low_rate
            (troe_a, T3, T1, T2?) ← falloff_coeffs (T2 default 0)
        else:
            (A,b,Ea) ← Arrhenius parameters

    return packed dict: {S,R,names,W,Tmid,a_hi,a_lo,v_f,v_b,reversible,
                        rxn_type,eff,A,b,Ea,A0,b0,Ea0,has_falloff,troe_a,T3,T1,T2}
```

B-1 Implemented chemistry mechanism ingestion pseudocode.

B.2 Thermodynamic closures (NASA-7) & T <-> e recovery

B-2 implementation of mixture properties in mass units and the robust Newton solve for **T from e** used inside the RHS.

```
# NASA-7 (per species)
(cp_k(T), h_k(T), s_k(T)) ← NASA-7(a_hi[k], a_lo[k], Tmid[k], T)

# Mixture internal energy (mass basis)
e_mix(T, Y) = Σ_k Y_k * u_k(T) / W_k      where u_k = h_k - R̄*T

# Mixture c_v (mass basis)
c_v, mix(T, Y) = Σ_k Y_k * [cp_k(T) - R̄] / W_k

# Temperature from internal energy (constant volume):
T_from_e(e_target, Y):
  bracket T ∈ [Tmin, Tmax]
  T ← improved initial guess from linearized e(T) around 1000-2000 K
  repeat (≤ max_iter) until |e(T) - e_target| ≤ tol:
    Δe ← e_target - e(T)
    ΔT_Newton ← Δe / max(c_v(T), ε)
    T ← clip( T + ΔT_Newton, [Tmin, Tmax] ), with bisection fallback for oversized steps
  return T
```

B-2 Implemented chemistry thermodynamic closures pseudocode.

B.3 Effective forward rates k_{eff} : elementary, third-body, falloff

B-3 assemble the pressure-dependent kinetics (third-body & Troe falloff) into a single vectorized call.

```
arrhenius(A, b, Ea, T) = A * T^b * exp(-Ea / (R̄ T))

forward_k_effective(T, C):
  k_inf ← arrhenius(A, b, Ea, T)
  M_r ← Σ_k eff[r,k] * C_k          # third-body collider
  k0 ← arrhenius(A0, b0, Ea0, T)    # low-pressure
  Pr ← (k0 * M_r) / clip(k_inf)     # reduced pressure
  F ← Troe_F(T, Pr, {a, T3, T1, T2}) # falloff factor
  return piecewise:
    if rxn_type == elementary: k_inf
    if rxn_type == 3rd-body: k_inf * M_r
    if rxn_type == falloff: k_inf * [Pr/(1+Pr)] * F
```

B-3 Implemented chemistry third-body falloff pseudocode.

B.4 Rates of progress & species production

B-4 numerically stable evaluation of R_r , $\dot{\omega}$ and heat release with log-products.

```

ROP_&wdot(T, Y, p, MECH, Kc_precomputed):
    # concentrations (clip to avoid zeros)
    C_k = clip(p * Y_k / W_k)

    # forward effective rates
    k_f = forward_k_effective(T, C, MECH)

    # precomputed equilibrium constants (Kc) from thermo (see §6.4.E)
    Kc = Kc_precomputed

    # mass-action products via log-sums for stability
    logC = log(max(C, ε))
    log Π_f_r = Σ_k v_f[r,k] * logC_k
    log Π_b_r = Σ_k v_b[r,k] * logC_k

    rop_f = k_f * exp(log Π_f_r)
    rop_r = (reversible ? (k_f / Kc) * exp(log Π_b_r) : 0)

    R_r = rop_f - rop_r
    wdot_k = Σ_r v_net[r,k] * R_r          # kmol/m³/s
    return wdot, rop_f, rop_r

```

B-4 Implemented chemistry species production rates pseudocode.

B.5 Precompute equilibrium constants $K_c(T)$.

B-5 avoid recomputing equilibrium constants each RHS call.

```

precompute_Kc(T):
    # species standard Gibbs: g_k(T) = h_k(T) - T s_k(T) (molar)
    g_k ← h_k(T) - T * s_k(T)
    v_net[r,*] = v_b[r,*] - v_f[r,*]
    ΔG_r = Σ_k v_net[r,k] * g_k

    K_p = exp( - ΔG_r / (R̄ T) )          # dimensionless
    Δv_r = Σ_k v_net[r,k]
    K_c = K_p * (p°/(R̄ T))^{Δv_r}       # to concentration basis
    return K_c

```

B-5 Implemented chemistry equilibrium constants calculation pseudocode.